Premios para los lectores • Estrategia • C y Ensamblador • Aventura • DIV 2 • Arcade

DIV manía

995 ptas.

PORTUGAL 990 ESC (CONT) 5,98 €

www.prensatecnica.com

Año 2 • Número 8

En el CD-Rom DEMOS
Painter3D
Poser 4
Canoma

• Acción plataformas
Aprende a programar saltos

• DIV Interno
Creando un generador de código

• Direct X
Destripamos Direct Sound v.7

• 3D
Nos prepararemos para la batalla

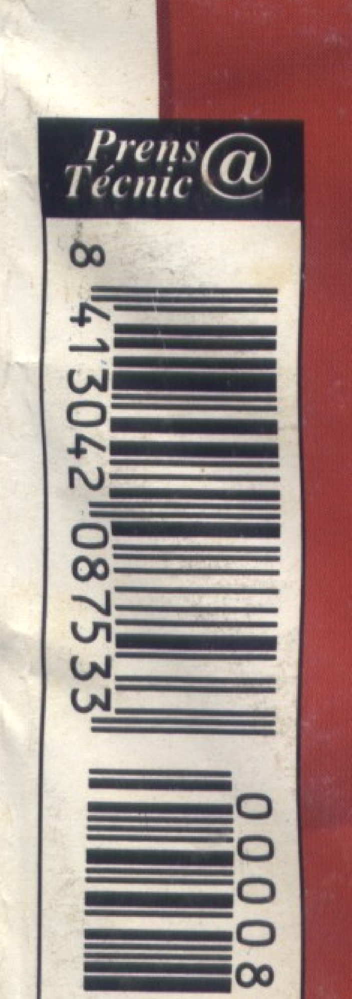
• Estrategia
Principios básicos de Inteligencia Artificial

• Especial virus
Clases de virus y su funcionamiento

• Poser
Haz que tus personajes se muevan

• Especial juegos
Realismo Vs. Jugabilidad





Programación de EMULADORES

Mil y un juegos en la pantalla de tu PC

work master



Web

Diseñar las mejores páginas web pasa por el conocimiento de los programas que en esta colección proponemos.



Publicidad

Trabajar como di-señador publicitario puede ser una realidad gracias a esta colección.

oque imagen
Convertir la
genes más si
en las m
ectaculare
una tarea se

Diseño y auto



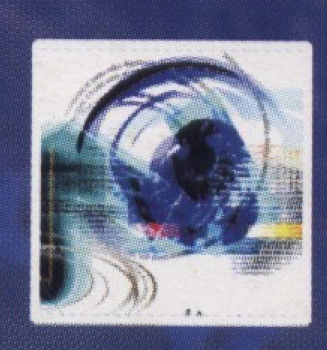
CorelDraw

Conoce a fondo este completo programa de dibujo que te permitirá realizar dibujos profesionales de la manera más sencilla e intuitiva.



Photoshop

Te enseñamos a utilizar de la manera más fácil y rentable el programa estrella del retoque fotográfico.



Freehand

Conoce todos los secretos de Freehand, el mejor programa de creación de gráficos vectoriales del mercado.

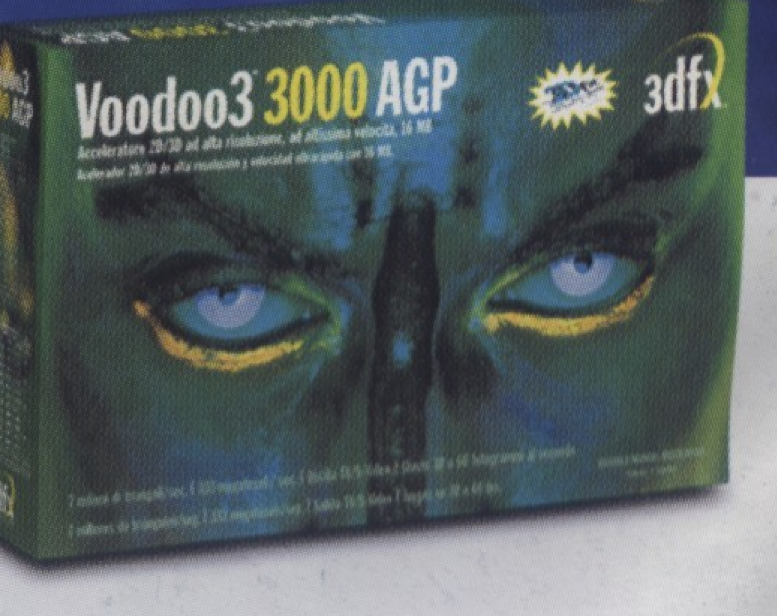


QuarkXPress

Cómo trabajar con QuarkXPress: el estándar de edición de los profesionales.



10 Tarjetas Voodoo3 3000 AGP retail





MacSchool



TeleTrabajo

Recomendado por:







Infórmate en el teléfono
91 304 06 22

Pre
Técn
de libros y

DIVmanía



¿Amas la programación de juegos...? Léete estas líneas

Saludos. Se acerca el verano y, una vez superados los exámenes (sabemos que muchos de vosotros sois estudiantes) por fin habrá tiempo para el ocio y el descanso. Esperamos que este tiempo sirva para que muchos de vosotros os pongáis manos a la obra y seáis capaces de crear magníficos juegos. En esta nuestra/vuestra revista hemos comprobado que el nivel de los Diveros aumenta cada día. Esperamos que siga así y que este tiempo de verano dé buenos frutos. Ah, y mucha suerte con los exámenes (un consejo de alguien que ya casi ni se acuerda de cuando acabó la carrera: estudiad y que la fuerza os acompañe).

Nada más, Divmanía sigue su curso con el interesante contenido de siempre y algunas novedades. Este mes tenemos una sección nueva dedicada a los juegos de acción y plataformas. Una idea que surgió de un reconocido DIVero y aquí está. Puede que muchos seáis escépticos pero en Divmanía tenemos las puertas abiertas a la participación de todos vosotros. Nos vemos dentro de dos meses.

Consejero Delegado: Mario Luis
mluis@prensatecnica.com
Director: Ricardo G. Bassols
Director Editorial: Eduardo Toribio
etoribio@prensatecnica.com
Director Editorial: José Henríquez
jhenriquez@prensatecnica.com
Director Técnico: Fernando Escudero
fescudero@prensatecnica.com
Coordinador Técnico: Oscar Condés
gover@prensatecnica.com
Dpto. de Redacción y Colaboradores: Alfredo del Barrio, Pablo Trinidad, Sergio Cánovas, Miguel Barroso, Emilio Llamas, Fermín Vicente, Ramón de España, Santiago Orgaz, Antonio Marchal, Santiago García, Jordi Martín, Daniel García y José A. Migens.
Dirección de Arte: Francisco Calero
Departamento de Maquetación: José A. Gil, Antonio García Tomé y Beatriz Fernández.
Portada: Jesús Mena
Publicidad: Marisa Fernández,
marisa@prensatecnica.com
Sonia Glez.-Villamil y Raúl Aguado.
Dpto. CD-Rom: Alvaro García y Enrique Roldán
Servicio Técnico CD-Rom: Auxiliadora Díaz
Horario de atención: tardes 16:00 - 18:00 h
E-mail: stecnico@prensatecnica.com
Secretaria de Redacción: Cori García Alonso
Departamento de Suscripciones: Sandra Fernández y Elena Fernández
suscripciones@prensatecnica.com
Departamento de Administración: Juan López, Juan José Martínez, Olga Peñas y Merche Quevedo
REDACCIÓN, PUBLICIDAD Y ADMINISTRACIÓN
c/ Alfonso Gómez 42. Nave 1.1.2
Madrid 28037. España
Tfno: (91) 304. 06. 22
Fax: (91) 304. 17. 97
Si llama desde fuera de España, marcar (+34)

Sumario

Programas del lector

PREMIAMOS LOS MEJORES JUEGOS DE LOS LECTORES

DIV ha creado toda una escuela dentro del mundo de la programación. Todos los que se han acercado a la herramienta han sido capaces de programar. Por ello realizamos un concurso entre los lectores que hayan realizado juegos con DIV. Os ofrecemos un primer premio de 25.000 pesetas y dos accésit de 20.000 pesetas.

¿QUÉ ES DIV GAMES STUDIO?

DIV Games Studio es una herramienta de programación que facilita en gran manera nuestra inmersión en el software de entretenimiento. Es el primer entorno profesional que permite realizar videojuegos con fines comerciales sin necesidad de un pago adicional. Con el carné de desarrollador incluido se permite el desarrollo de cualquier tipo de juegos y su libre venta y distribución.



E-mail: divmania@prensatecnica.com
http://www.prensatecnica.com
Horario de atención al público: de 09:00 a 19:00 h ininterrumpidamente

EDITA: PRENSA TÉCNICA
Consejero Delegado: Mario Luis
Director General: Ricardo G. Bassols
Directores Editoriales: Eduardo Toribio y José Henríquez

Director Técnico: Fernando Escudero
Director de Producción: C.P. Cerezo
Directora Publicidad: Marisa Fernández
Director Financiero: Joaquín González
Fotomecánica: Prensa Técnica
Impresión: Printerman

Duplicación del CD-Rom: M.P. O., Servicios Ibéricos, Grupo Cóndor
Distribución: SGEL. Avda Valdelaparra, 29 Alcobendas. Madrid

Depósito legal: M-42077-1998
AÑO 2 • NÚMERO 8
Copyright 30-09-00 – PRINTED IN SPAIN

La moral del programador

Tenéis la palabra

Saludos y bienvenidos a un número más de nuestra/vuestra revista. Este mes dedicamos esta sección a un interesante mensaje que nos ha llegado y que incluye un programa que encontraréis en el Cd. La palabra es vuestra.

Moral?! ... pero, ¿existe eso en programación? Efectivamente, existe. Y lo que trataré de hacer será concienciar a los lectores programadores de que existe una moral, que, a pesar de no ser muy conocida, debiera de estar presente en todas las personas a la hora de programar, a pesar de que estas letras sean el mero vehículo para hacer que los lectores discurran sobre el tema.

Se me ocurrió escribir sobre el tema al sentirme inspirado por la frase de Ferminho que aparece en DIVmanía nº7: "**...el código de los juegos puede optimizarse en la mayoría de los casos...**", de la que se puede extraer el jugoso tema.

¿A qué me refiero con la moral? Pues la moral, según mi punto de vista, se puede definir como la forma de crear un programa, o mejor dicho, la filosofía con la que está programado, en este caso, el juego. Un juego puede estar creado de tres formas: pensando en la comodidad a la hora de programar, pensando en la velocidad del juego, y pensando en los recursos del jugador.



La comodidad concierne al programador de forma que es éste el que, por comodidad a la hora de programar, prefiere tener menor velocidad y/o recursos a cambio de serle más fácil el desarrollar un juego. Ejemplos de esta posición pueden ser programaciones visuales, en las que prima la comodidad, y también podemos poner de ejemplo el DIV.

La velocidad del juego es una de las cosas que se está olvidando desde hace no mucho tiempo. Sin duda, para conseguir la mayor velocidad lo mejor sería programar el juego completo en ensamblador, aunque, realmente, no vale la pena, pues el tiempo necesario para desarrollar el juego sería demasiado, pero con esto quiero decir que los programadores deben de pensar en los mejores algoritmos para realizar una tarea.

La última forma está en pensar en los recursos del jugador. Este caso puede decirse que es una derivación de la comodidad del programador, puesto que es más fácil y cómodo tener absolutamente todos los gráficos de un juego en un .fpg y no distribuirlos en varios según los vaya a necesitar.

Por supuesto, es imposible encuadrarse en una de estas tres formas de programación, pero lo que sí hay que pensar es que "**la virtud está en el equilibrio**", y que, en muchos casos, tanto la velocidad como los recursos están olvidados, siendo aún mayor el olvido en este último, puesto que, según parece, hay juegos que no funcionan en ordenadores con 16 Mb de RAM.

Propongo que se piense más en la velocidad y los recursos, de forma que se distribuyan las imágenes en varios ficheros, o que se programe una configuración aparte para que no se carguen determinados sonidos, o que se pierda un poco de calidad repitiendo imágenes para ahorrar memoria. Para este fin, existe mi programa DIVRC.EXE , el cual crea el fichero DIVRC.INF que se puede leer y así saber el nivel de recursos a cargar.

Podemos tomar como ejemplo el código de los diálogos de la DIVmanía nº7 (página 47), en la que un programador que pensase más en la comodidad, programaría una función que re recortase las frases, de forma que no tendría que introducir las frases al contrario, y, seguramente, este cambio no afectaría ni a los recursos ni, seguramente, a la velocidad. Con esto quiero demostrar que buscar algo de comodidad no es algo negativo, puesto que esto nos permite desarrollar las cosas más rápidamente.

NiShi

Avalancha de segundas partes





La feria de videojuegos más importante, la E3, ha clausurado sus puertas. Casi todos los desarrolladores del planeta usan este certamen como escaparate de sus trabajos y parece que este año les ha dado por las segundas partes, sólo tenéis que echar un vistazo a esta lista:

Alundra 2, Armored Core 2, Baldur's Gate 2, Call to Power 2, Commandos 2, Croc 2, Dark Reign 2, Dead or Alive 2, Diablo 2, Dino Crisis 2, Driver 2, Dynasty Warriors 2, Evolution 2, Frogger 2, Grandia 2, Independence War 2, Marvel vs. Capcom 2, MechComander 2, Medievil 2, Mega Man Legends 2, Midtown Madness 2, Motocross Madness 2, Nightmare Creatures 2, Power Stone 2, Ready to Rumble 2, Runabout 2, Sonic 2, Soul Reaver 2, Street Skater 2, Strider 2, Syphon Filter 2, Team Fortress 2, Tenchu 2, Tony hawks Pro Skater 2, Tribes 2, Wild Arms 2, etc., y eso que no contamos con las terceras o cuartas partes de más títulos famosos, como por ejemplo la tercera parte de *Warcraft* o la cuarta entrega de *Monkey Island*, si lo hiciésemos la lista sería interminable.

Mención aparte merecen también el montón de juegos deportivos que llevarán como coletilla el año 2000 ó el año 2001 como apellido añadido a su nombre original:

All Stars baseball 2001, FIFA 2001, Formula one 2000, International Track and Field 2000, Jeremy Mc Grath Supercorss 2000,Knockout Kings 2001, Madden NFL 2001, Nascar 2001, NBA 2K1, NBA Live 2001, NBA Showtime 2001, NCA GameBreaker 2001, NCAA Footbal 2001, NFL 2K1, NFL Blitz 2001, NFL Gameday 2001, NFL QB Club 2001, NHL 2001, Sammy Sosa Baseball 2001, Superbike 2000, Sydney 2000, Triple Play 2001,World Series Baseball 2K1, etc.





¿Impresionante, verdad? bueno no todos los desarrolladores optan por programar juegos basados en títulos antiguos, algunos incluso usan la cabeza para presentar nuevas propuestas, ese es el caso de Matrix o el juego inspirado en la película de "El Planeta de los Simios".



Nueva versión de Fallout en el E3

Quien sea aficionado al rol conocerá de sobra este juego cuyas dos partes fueron desarrolladas por los estudios Black Isle. Ahora conocerá una transformación total en su modo de juego de la mano de la compañía 14 Degrees East, también de Interplay, pasará a llamarse *Fallout Tactics* y será del género de la estrategia con combate por turnos, con una mecánica parecida a *Jagged Alliance 2*; en cuanto a la estética no habrá demasiados cambios, seguiremos inmersos en un apocalíptico y devastado mundo post-nuclear con personajes sacados de una película de Mad Max, de los treinta disponibles se podrán elegir a seis de ellos para jugar una de las 20 misiones, con seis etapas cada una, con las que contará el juego.



El código de Falcon 4 en Internet

Alguien desconocido liberó hace unos días el código fuente completo del simulador de vuelo Falcon 4 de la compañía Hasbro. Cuando en esta empresa se enteraron de la noticia saltaron las alarmas e intentaron limpiar Internet de cualquier resto de código. Como podréis imaginar no lo consiguieron, hoy por hoy puede ser encontrado si se busca en los sitios adecuados.

Aunque Hasbro ha prohibido la distribución de su trabajo, difícil lo tiene ya para que este código no sea usado por los aficionados a la programación y a los simuladores de vuelo para hacer las modificaciones que quieran en el juego original.

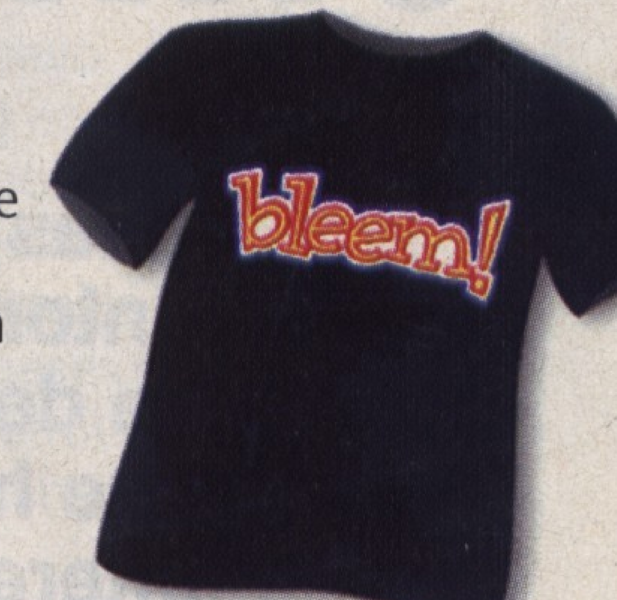


Dreamcast en el punto de mira de Bleemen

Como ya todos sabréis a estas alturas, Bleem, la compañía que ha sido objeto de múltiples querellas judiciales por parte de Sony con motivo de la distribución de un emulador que permite jugar con juegos de PlayStation desde un PC, ha anunciado que la consola de Sega, Dreamcast, también contará próximamente con un emulador que permitirá probar buena parte de los juegos diseñados para Play en la plataforma Dreamcast.

El nuevo producto ya ha sido presentado en el E3 y se llamará BleemDC. La comercialización de este nuevo producto se hará a lo grande: habrá cuatro versiones del emulador distintas, cada una de ellas compatibilizará el uso de unos 100 juegos de PlayStation en la Dreamcast; además sacarán a la venta periféricos propios, un ejemplo es un mando Dual Shock que servirá para jugar cómodamente desde la consola Dreamcast a los juegos de PlayStation.

No creemos que los directivos de Sega tengan problema alguno con el nuevo programa, incluso servirá para que la demanda de consolas Dreamcast aumente, pero los representantes de Sony deben estar que muerden, no han sido capaces de parar judicialmente la extensión del programa Bleem!, como éste no utiliza la bios de Sony sino que emula el manejo de la consola a través de rutinas de programación propias, ningún juez ha dado la razón a la compañía nipona e incluso tenían la orden judicial de que ninguno de sus trabajadores se acercase al stand de Bleem en el pasado E3 ya que en edición anterior de la feria ambos contendientes se cruzaron algo más que palabras y demandas judiciales.



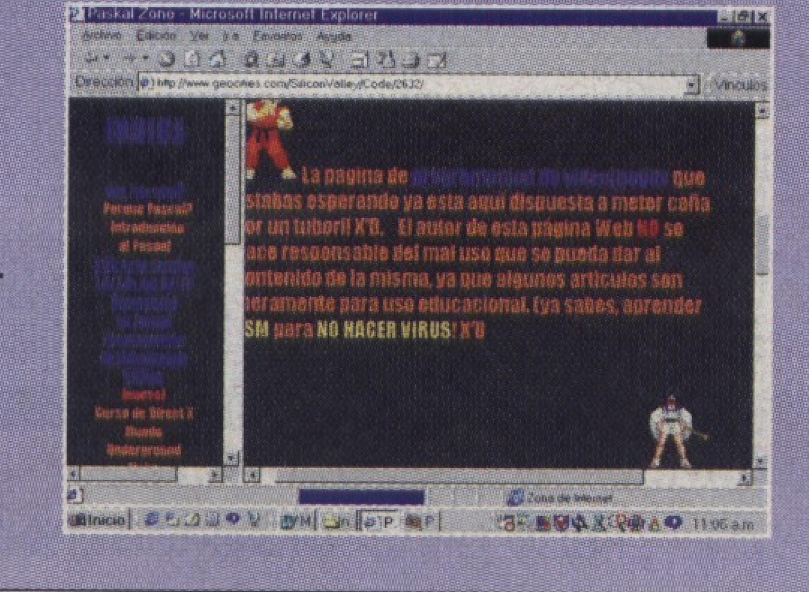
Vital Web cambia de servidor

La dirección de la página que destripábamos en un número pasado en la sección Web Div de la revista, la publicación electrónica Vital Web, ha cambiado de lugar, aunque inicialmente la url era http://www.geocities.com/timessquare/arcade/9520, actualmente está situada en el servidor de pago http://www.electronic-soft/vitalweb/.

También se puede acceder a ella a través de los siguientes redireccionadores:

http://pagina.de/vital
http://pagina.de/vital-web

La página ha sido totalmente actualizada por lo que es aconsejable que te pases por allí para echarla un vistazo.



Vital también nos ha hecho llegar unas cuantas direcciones de páginas que tienen que ver con Div y la programación de juegos, así que vamos a compartirlas con vosotros:

http://pagina.de/tyrion
http://personales.com/venezuela/valencia/eichgame
http://pagina.de/slainte
http://pagina.de/pandoria (nueva page de Deemo)
http://divgames.com (actualizada por Roman1)
ftp.xoom.com (nuevo ftp DiV)
http://stratos-ad.com

Gracias por tenernos al día Vital, esperamos seguir teniendo noticias tuyas, y si alguno tiene algo que contarnos o alguna información que compartir con el resto de diveros, que no dude en escribirnos.

El regreso de Guybrush Threepwood

Cuando el río suena es porque agua lleva, por lo menos eso dicen, y en el caso de la especulación con la posible cuarta parte de *Monkey Island*, más que río es torrente tumultuoso por la cantidad de noticias relacionadas con este conocidísimo juego que han inundado la red en cuanto se abrió el grifo de los rumores.

Hace poco apareció una encuesta en la revista electrónica Meristation preguntando a los internautas cuál era el título con el que primero habían jugado o el primero que recordaban. Muchas de las respuestas citaban a *Monkey Island* como un juego inolvidable y añorado por todos aquellos que empezaron a iniciarse en este vicio hace la friolera de 10 años.





Pues bien la cuarta parte de esta obra ha sido ya oficialmente anunciada, llevará por título *Escape from Monkey Island* y será, cómo no, una aventura gráfica, la novedad es que contará con gráficos totalmente tridimensionales esta vez, posiblemente realizados con el motor gráfico de *Grim Fandango*. Esperemos que mantenga el estupendo humor que atesoraban las anteriores entregas. ¡Cómo añoramos a Guybrush!

Una compañía muy original
Shiny Entertainment

¿Has visto algo parecido alguna vez? ¿No? Entonces es de Shiny. Esa podría ser la forma de describir la filosofía de esta desarrolladora que ha creado algunos de los juegos más interesantes de los últimos tiempos. Entre ellos, Messiah, el título que ha hecho que todas las miradas se dirijan hacia Shiny.

El pasado año, en octubre para ser precisos, celebraron su 5º aniversario como compañía. Para ellos, la fecha no fue solamente una reflexión sobre el pasado sino también una mirada al futuro. Evidentemente, nosotros no podemos hablar de su futuro (si acaso de sus proyectos más inmediatos) pero sí que podemos dar un pequeño repaso a la historia para ver qué es lo que han hecho los chicos de Shiny por el mundo de los videojuegos.

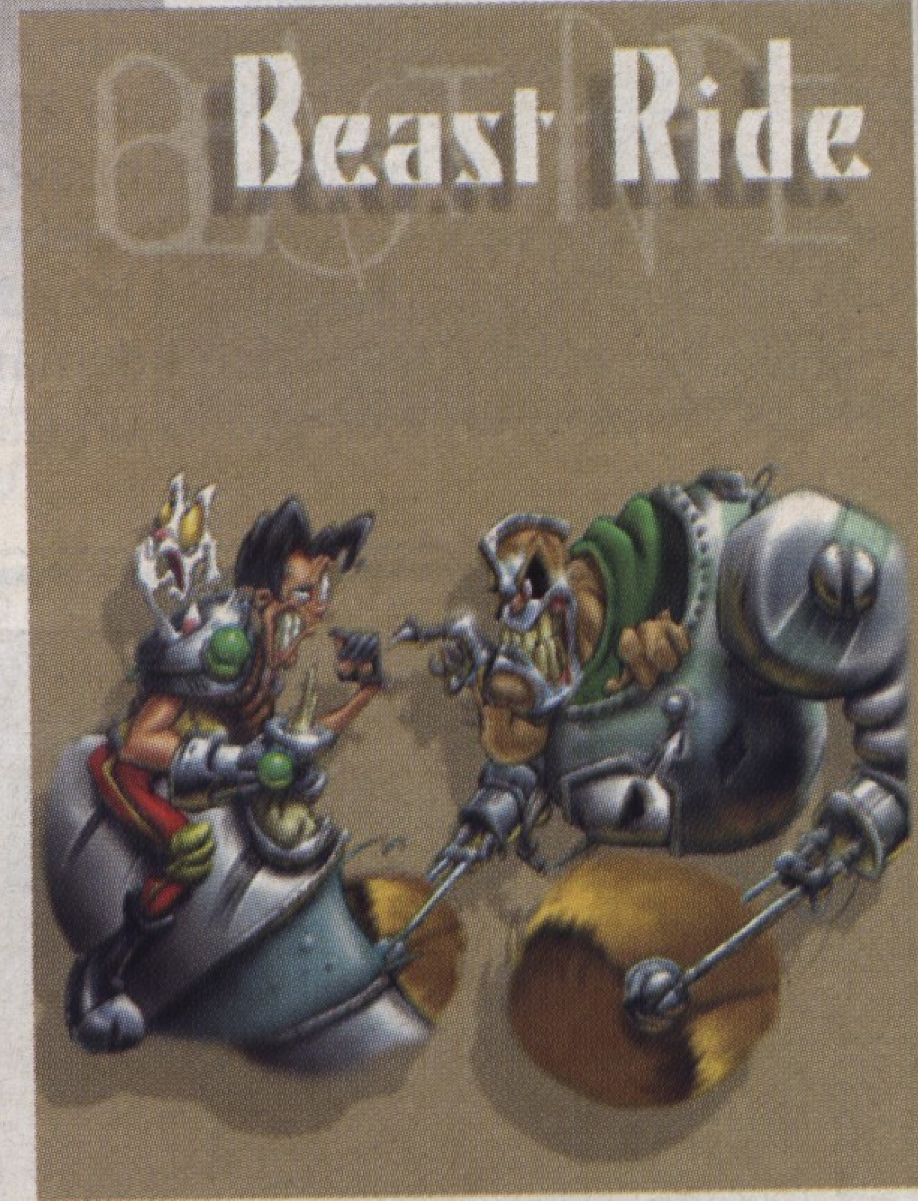
David Perry

No sólo es el presidente y cabeza visible de la empresa, sino su genio creador. Tiene 33 años, nació en el norte de Irlanda, y ahora vive en una playa del sur de California. Ama los juegos, aunque odia los que están llenos de vídeos y pantallas de carga. Le gustan las películas de acción, tipo *Aliens, Predator*, etc. y la música dance, U2 y el folklore irlandés, que escucha mientras trabaja. Practica muchos deportes, entre ellos el submarinismo y el vuelo en helicóptero.



Perry tiene un sexto sentido para los juegos, un talento innegable para saber qué es lo que va a funcionar a nivel mundial. Los personajes de sus juegos, sin duda uno de los puntos fuertes de sus productos, han traspasado la frontera del software para pasar al mundo del espectáculo en todas sus dimensiones: televisión, el cine, los cómics...

Considerado una de las figuras más importantes del mundo de los videojuegos, su nombre ha dejado una huella imborrable en el mundo del soft lúdico desde los tiempos del Spectrum hasta los del Pc en 3D. Su nombre es sinónimo de calidad y diversión.



Shiny Entertainment y David Perry tienen una vasta experiencia construida a través de los años desde que en 1981 David pusiera sus ojos en una “máquina calculadora de bolsillo”, la Sinclair ZX81 y se quedara “alucinado” viendo como un punto que simulaba una pelota iba de un lado a otro de la pantalla en la que había unos palos que hacían las veces de raquetas de tenis. Fue el comienzo de una obsesión que le llevó, cuatro años después a crear su primer juego para Spectrum: Pyjamarama.

Eso sólo fue el comienzo de una carrera meteórica que le llevó a trabajar en muy diversos proyectos hasta que decidió crear su propia compañía. Virgin, la empresa para la que trabajaba no se alegró precisamente de su marcha pero David tenía que cumplir un sueño. De modo que se puso manos a la obra y consiguió reunir algunos de las mentes más creativas del mundo de los videojuegos







para hacer un grupo de gente que convirtiese a Shiny en una empresa potente, una de las más influyentes dentro del panorama del software lúdico.

Salto a la fama

Perry aún recuerda cuando ellos eran una pequeña empresa que quería hacer juegos para Sega. Recuerda especialmente cierta reunión con ejecutivos de esta empresa en la que le soltaron todo un rollo sobre el mercado de los videojuegos, su posición de guardianes del mercado y la de Shiny como pequeña compañía de "bichos raros". David cuenta como una vez aguantado el "discurso" pudo mostrar lo que habían hecho con su proyecto EarthWorth Jim y como la sala se quedó muda y los altos ejecutivos se quedaron estupefactos ante lo que vieron. Quedaron tan impresionados que en unos minutos tenían un contrato, la portada de una revista de Sega y el respeto de las mejores compañías de juegos del mundo.

EartWorth Jim conmocionó la comunidad de los videojuegos. Con su animación fluida y su curioso sentido del humor consiguieron atrapar el interés del gigante Sega, primero, y del resto del mundo después. El juego tuvo tal éxito que no solo se produjo una secuela sino que derivó en la venta de tres licencias al margen del mundo del software lúdico. Unas licencias que resultaron muy lucrativas, como la línea de juguetes y figuras animadas que rivalizó con las mismísimas tortugas Ninja. Su popularidad creció hasta el nivel de personajes legendarios como Mario Bross o Sonic. Pero eso sólo fue la punta del iceberg. En 1994, Jim, el personaje protagonista, tuvo su propia serie de TV y sus propios cómics con la firma de Marvel. El éxito fue tal que la pequeña empresa de Laguna entró a formar parte del equipo de uno de las desarrolladoras internacionales de más prestigio: Interplay.

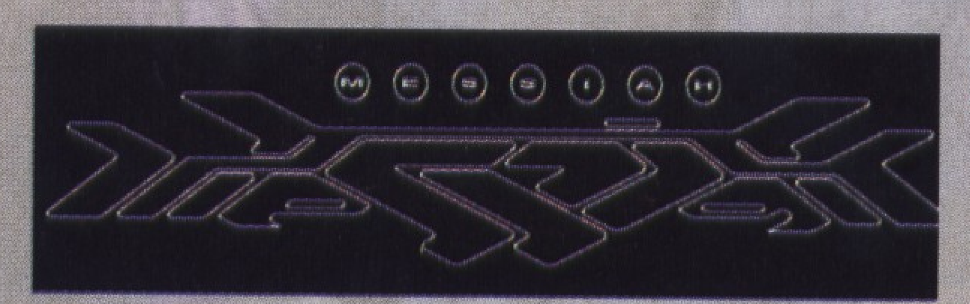
En el año 96, David Perry y sus chicos, lejos de quedarse en un solo éxito, volvieron a la carga con una historia protagonizada por un científico loco, un perro mutante y un joven con un arma por cabeza...

MDK

Efectivamente, se trata de los personajes del no menos famoso MDK, el más ambicioso proyecto de la compañía. Ellos conocían el mercado de las consolas mejor que nadie y su éxito con Eartworth Jim 3D era más que considerable, sin embargo, el mercado de PC era nuevo para ellos. Un mercado impredecible, saturado de productos y con "muy mala prensa".

Pero Shiny Entertainment tenía algo que probar. Un fallo les hubiera cerrado en las narices el mercado de Pc para siempre, pero de nuevo acertaron. MDK se convirtió en uno de los juegos de Pc más famosos de todos los tiempos vendiendo la increíble cantidad de medio millón de copias en apenas unos meses del lanzamiento. La prensa, generalmente muy reservada, se deshizo en elogios y premios.

En la actualidad, Shiny Entertainment se ha convertido en una empresa fuerte. La empresa negocia licencias para llevar al cine y la televisión las aventuras de MDK y trabaja constantemente en la cre-



Messiah

Un título de "campanillas" que ha hecho centrar toda la atención en los chicos de Shiny. En él, nos encarnaremos en Bob, un angelito enviado por el Sumo Hacedor para poner orden en un mundo degenerado y corrompido hasta límites insospechados, tiene cierta facilidad para poseer el cuerpo de cualquiera que se le cruce en el camino, ya sea persona, animal u organismo biológico.

La mecánica para lograrlo es sencilla. Por ejemplo, si vemos un soldado que queremos poseer nos situaremos cerca de él y saltaremos sobre él. Cuando Bob esté dentro de su cuerpo podremos manejarlo como si de nosotros mismos se tratase, usar sus armas contra otros personajes del juego, realizar determinadas acciones, o suicidar a nuestro envoltorio carnal.

Con esto la variedad está asegurada. El juego permitirá actuar como una buena persona o como un malvado que sembrará semillas de destrucción allá por donde deambule. El escenario donde transcurre la acción es una metrópoli futurista llena de adversarios astutos y trampas diabólicas.

No podemos pasar por alto la mención de la alta tecnología con la que contará este título. La empresa Shiny se está convirtiendo en una fuente de desarrollo de todo tipo de técnicas aplicables a los videojuegos. En este caso han creado el sistema RT-DAT (mosaico y deformación en tiempo real). La deformación en tiempo real crea personajes en el juego con todos los rasgos físicos de un ser humano. Cada personaje de *Messiah* tendrá una estructura ósea real, con músculos que unan estos huesos y una piel a modo de textura que recubra al personaje.



Por otro lado, la tecnología empleada se adaptará a las capacidades del ordenador en que se instale. Una gran idea.





Otros juegos



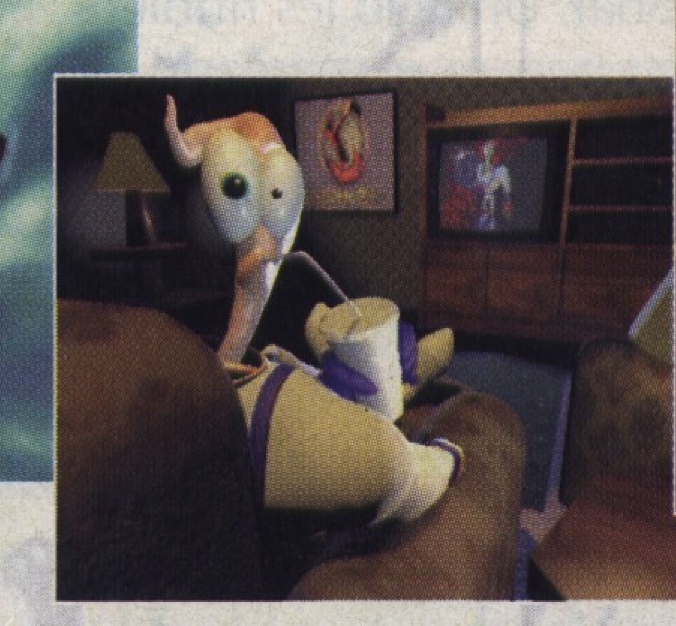
Tras el bombazo de Messiah, Shiny ya tiene preparado su nuevo producto estrella. Se trata de *Sacrifice*, un juego totalmente tridimensional enclavado en el género de la estrategia en tiempo real.

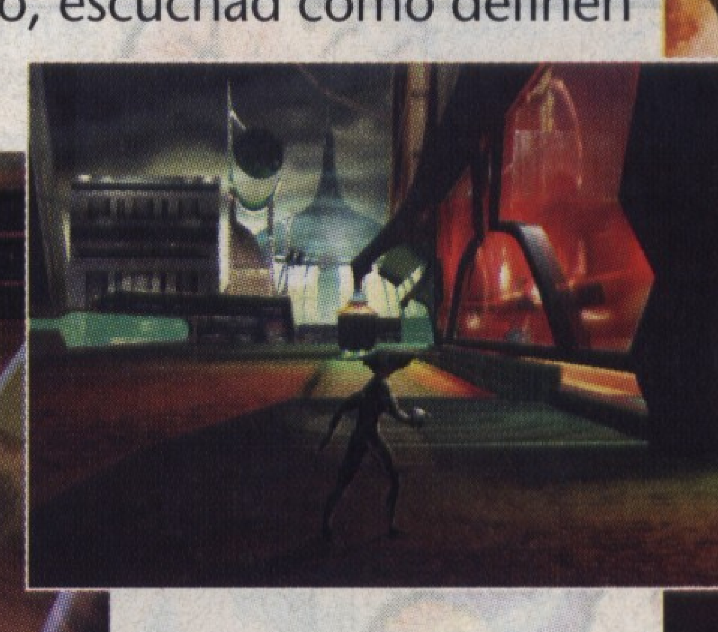
No sabemos mucho sobre este juego, pero sí lo suficiente como para estar desde ahora pendientes de cualquier noticia relacionada con él. Parece ser que el juego será del género estratégico pero con entornos tridimensionales. Muchas unidades para manejar, auténticas legiones de todo tipo de elementos, cada uno con funciones distintas, y controlados a través de la división de la pantalla en varias ventanas.

Una novedad con la que contará este juego es un nuevo tipo de interfaz inteligente. Por lo visto podremos memorizar en el programa determinados movimientos del ratón, los que nosotros queramos o nos sean más fáciles de realizar. Cuando lo hayamos hecho no tendremos que pulsar los botones del periférico para impartir ordenes a diestro y siniestro. De este modo nos ahorraremos mucho tiempo.

Otros proyectos menos conocidos de Shiny son títulos como *Stunt Copter*, un curioso título basado en helicópteros de radio control y *Wild 9* una especie de juego de rol caracterizado por tener mucha acción, aventura y un humor salvaje; un innovador sistema de juego y unos personajes realmente curiosos. Y si no, escuchad cómo definen ellos mismos el juego: imagínate que George Lucas hubiera co-escrito *Star Wars* con Lewis Carroll, el autor de Alicia en el país de las Maravillas, y la hubiera co-dirigido con Tex Avery, el genial creador de cartoons. Pues eso es *Wild 9*.











ación de los juegos más avanzados técnicamente, especialmente en juegos para la próxima generación de consolas.

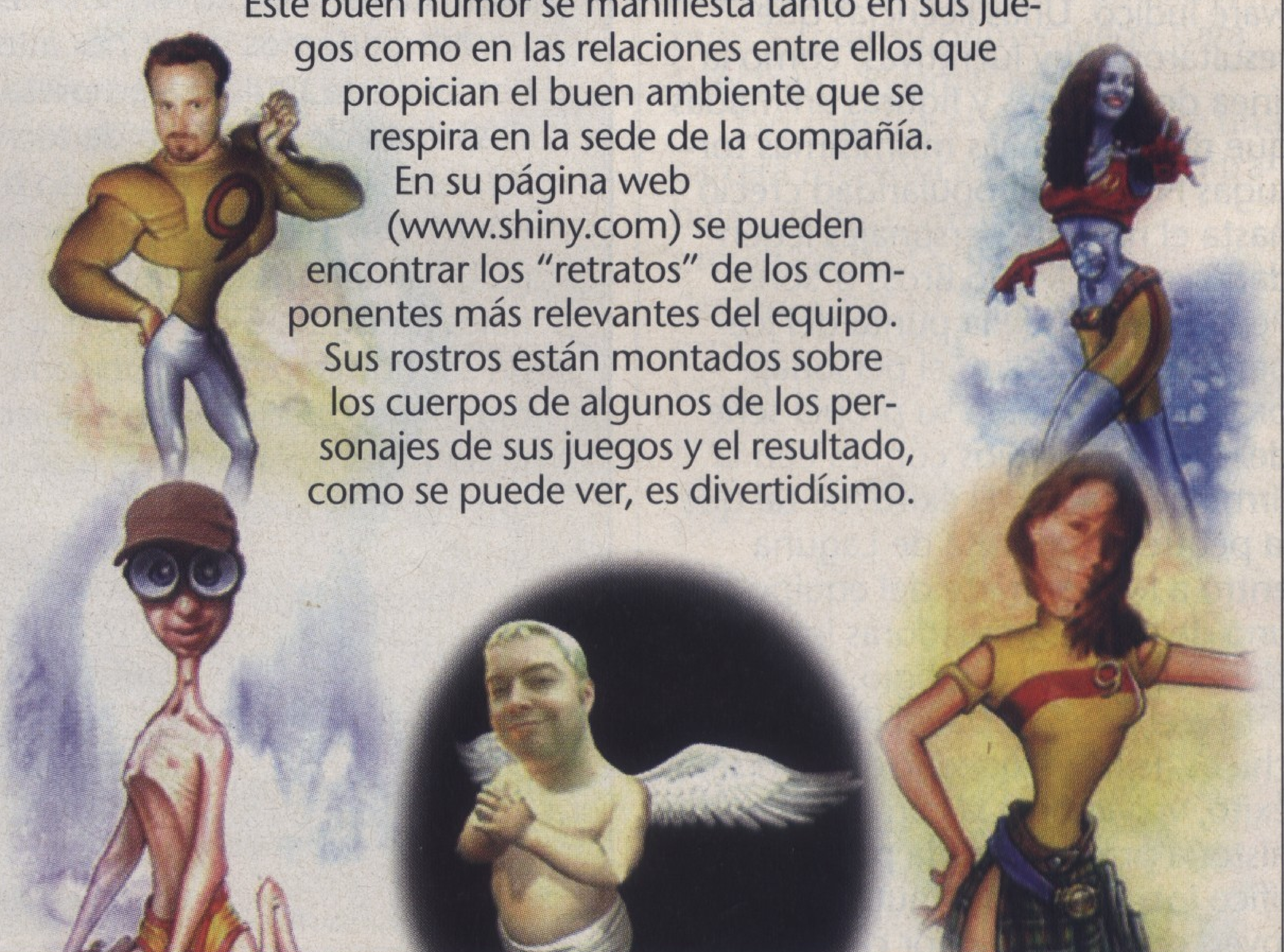
David Perry aparece regularmente en shows de televisión, noticias y da charlas al resto de la industria. Recientemente fue nombrado por la revista *Next Generation* como el hombre más importante de la industria del videojuego. Perry es mucho más que un simple programador. Es un hombre de negocios con un sentido natural para éstos. No está mal para un chico de County Antrim, Irlanda del Norte, que empezó con un Sinclair ZX81, un mal corte de pelo y una gran pasión por los videojuegos, en una industria que muchos creyeron que no sería más que una moda pasajera.

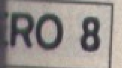
"Tenemos que estar al día y ser constantemente innovadores y comercialmente aceptables si no queremos ser destruidos por empresas de todo el mundo". Esa puede ser la traducción de su filosofía. Una filosofía que le ha llevado al éxito. Desde luego, la innovación es fundamental y el humor imprescindible; eso se ha demostrado con creces.

Oscar Condés

Ante todo mucho humor

El buen humor es la mejor característica de una empresa joven formada por personas cuyas mejores características son el talento y la creatividad. Este buen humor se manifiesta tanto en sus juegos como en las relaciones entre ellos que propician el buen ambiente que se respira en la sede de la compañía. En su página web (www.shiny.com) se pueden encontrar los "retratos" de los componentes más relevantes del equipo. Sus rostros están montados sobre los cuerpos de algunos de los personajes de sus juegos y el resultado, como se puede ver, es divertidísimo.





Las mil y una máquinas en tu PC

Emuladores: conceptos y nociones de programación

Este mes hemos dedicado nuestro reportaje al apasionante mundo de los emuladores, un campo muy en boga en los últimos tiempos. Nuestra intención es no sólo contaros qué son y cómo funcionan sino introduciros en las claves de su programación.

Desde hace algún tiempo, cuando alguien navega por Internet buscando archivos que "bajarse" tiene una nueva y adictiva alternativa a las imágenes o los archivos MP3. La Red está ocupada por una horda de nostálgicos de la informática en busca de aquel fragmento de software que tantas horas de sus vidas llenó, ya sea en la tranquilidad de sus cuartos cuando un obsoleto ordenador de 8 bits hacía sus delicias, o bien en la típica sala de recreativos en compañía de la gente habitual del barrio. Si no es la nostalgia la que mueve a estos internautas, es un interés de coleccionista o la avidez del descubridor de diferentes sistemas informáticos. Estamos hablando de los emuladores: un nuevo universo de posibilidades para nuestro PC.

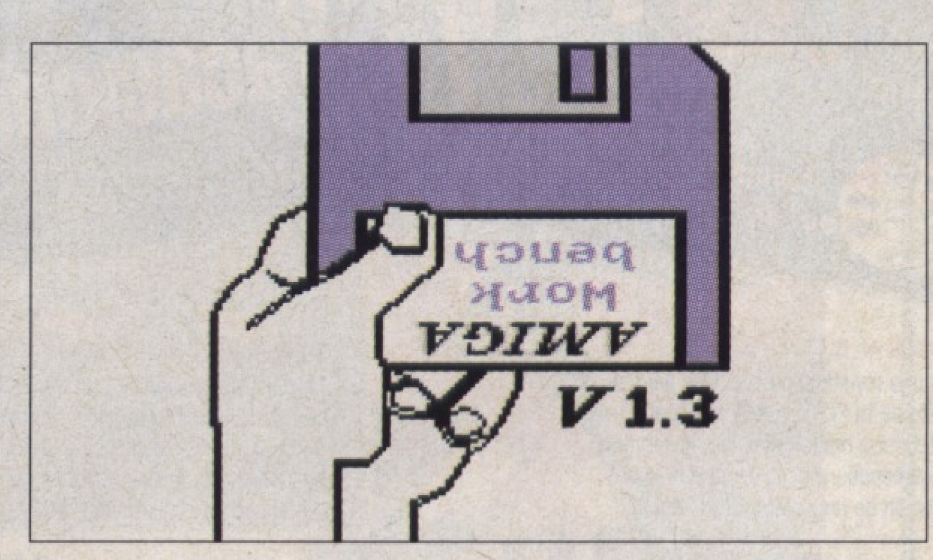
Gracias a los emuladores podremos volver a utilizar programas del pasado en nuestro PC

¿Qué es un emulador?

Si buscamos en el diccionario la voz *emulador* encontramos la siguiente definición: "Que emula o compite con otro" y en *emular* encontramos "Imitar las acciones de otro procurando igualarlas e incluso excederlas". Si trasladamos esta definición al campo informático un emulador es un programa cuyo cometido es el de hacer funcionar software escrito para otros sistemas en nuestro PC. De forma más clara, un emulador sirve para ejecutar programas de otras máquinas distintas, desde Macintosh hasta el anciano Spectrum, pasando por todo tipo de consolas. Incluso puede ser emulado en el PC el comportamiento de chips aislados, como el famoso Z80, o incluso un coprocesador matemático. En gran parte de los casos, como se trata de máquinas de tecnología anterior a la de los PCs actuales, se consigue igualar sin problemas el funcionamiento de estas máquinas y, en muchos casos, mejorarlas, aunque en casos como los de las videoconsolas de última generación no se consigue una emulación tan perfecta.



Pantalla de presentación del juego AMC en una emulación de CPC6128.



Pantalla de presentación de la ROM de AMIGA en un emulador.

¿Qué interés puede tener emular una máquina distinta de la nuestra?

Son muchas las motivaciones que pueden inducir a la programación de emuladores. Quizá la más antigua sea la basada en intereses educativos: por ejemplo, si en una facultad de informática se desea que los alumnos realicen prácticas en lenguaje ensamblador de procesadores de la familia del Motorola 68000, se tendrían dos opciones. Una es la de comprar el número suficiente de ordenadores Macintosh para que todos los alumnos puedan realizar dichas prácticas, y la otra, mucho más barata, es la de desarrollar un emulador por software que sea capaz de traducir instrucciones de ensamblador del 68000 a instrucciones del Intel x86 utilizado en los PCs. Claramente, la segunda opción parece más viable.

Otra de las posibles motivaciones puede ser la antes mencionada nostalgia. Muchos de los actuales usuarios de PC hemos sido en el pasado usuarios de otros sistemas que se ganaron nuestro aprecio después de innumerables horas de uso (ya se sabe que el roce hace el,



Emulación de recreativas de última generación con Callus (King of Dragons).

cariño) y, en bastantes ocasiones, hemos sentido el deseo de conectar nuestro viejo ordenador para pasar un último rato disfrutando con el que era nuestro juego favorito. Disponiendo de un emulador podremos sentir las mismas sensaciones de antaño haciendo solamente doble clic sobre el icono correspondiente en nuestro sistema operativo Windows.

Otra razón puede ser tener en nuestro disco duro todo el software disponible para una máquina, sobre todo, cuando el existente resulte de difícil acceso. En el caso de consolas de videojuegos que ya no estén en venta, puede que esta sea la única forma de conseguir determinado software. Además, hay que considerar que para almacenar todos los cartuchos editados para una consola necesitaríamos disponer de unos cuantos metros cuadrados extra, mientras que con un emulador bastará con disponer de unos pocos megas en nuestro disco duro, aunque, como comentaremos después, no está tan claro que se pueda hacer esto.

Por último, puede ser que lo que nos impulse a programar un emulador sea el propio reto que ello representa, ya que la programación de un emulador es un proyecto nada trivial y dotado de una considerable complejidad. A continuación, después de definir un concepto importante, veremos con más detalle los distintos tipos de emulador existentes, lo cual nos ayudará a comprender mejor a lo que nos enfrentamos antes de adentrarnos en los secretos de su programación.

ROMS: la parte turbia de este interesante mundo

Si nos adentramos en una de las numerosas páginas web que tratan del tema que nos ocupa, veremos que ofrecen la posibilidad de "bajar" gran número de archivos, llamados ROMS, sin los cuales los emuladores carecerían de sentido, pero cuyo uso, se nos advierte continuamente en estos web sites, no es libre, indicando que las ROMS están incluidas sólo a modo de ejemplo y que deben ser borradas después de haber probado el emulador.

Primero, debemos saber qué se entiende por ROM: las siglas ROM corresponden a los términos ingleses *read only memory*, memoria de sólo lectura. Ejemplos de estas ROMS pueden ser la BIOS de los PCs que contienen la información básica que el ordenador necesita para funcionar en ausencia de todo software (sin ellas no serviría de nada introducir un disquete de arranque para comenzar a instalar un sistema operativo en un ordenador nuevo, por ejemplo). En el caso de las consolas de videojuegos se les llama ROMS a las imágenes de los cartuchos que contenían originalmente el software, es decir, si no tenemos ROMS, no tenemos juegos que correr en nuestro ordenador.

Cómo parece lógico, estas ROMS han sido obtenidas a partir de software comercial, sujeto a leyes de copyright, y su uso es ilegal. De ahí que en las páginas web relacionadas con el tema se ofrezcan estas ROMS, pero se delegue la responsabilidad de su uso incorrecto al usuario.

Emuladores de procesadores

Los vemos en primer lugar por ser los más sencillos de programar, ya que de estos emuladores sólo se espera, en principio, que muestren el estado de los distintos registros del mismo al ejecutar una o varias instrucciones propias. Esto significa que no se necesita programar interfaces gráficas complicadas, ni emular sonido, ni nada por el estilo. Aunque estos proyectos resulten a priori menos ambiciosos, son fundamentales para la realización posterior de emuladores de máquinas complejas, ya que esas máquinas estarán compuestas de algunos de estos procesadores. Cuando se programa este tipo de emuladores con el fin de incorporarlos en proyectos de mayor envergadura reciben el nombre de *engines* o motores, cuya importancia trataremos más adelante.

Emuladores de consola

Estos emuladores se basan en *engines* de procesadores como el Motorola 68000 o el Z80. En el caso de estos emuladores, su programación es ya bastante más compleja ya que nos tenemos que preocupar de cosas tales como los gráficos o el sonido. Un factor que



Callus, además de ejecutarse en ventana, aplica suavizado a la imagen.

determinará la dificultad de la programación de uno de estos emuladores será la generación a la que pertenezca la consola en cuestión, es decir, su antigüedad. Cuanto más moderna sea la consola será más complicado emularla con eficacia.

¿Y cómo saber con antelación si nuestro PC tendrá potencia suficiente para emular una consola determinada? La verdad es que no resulta fácil porque hay diferencias sustanciales entre un ordenador y una consola: en un ordenador todo el peso del proceso lo lleva la CPU, que suele ser mucho más potente, en proporción, que la CPU de una consola. Sin embargo, en el caso de la consola, la CPU se ve ayudada por un conjunto de coprocesadores especializados en gráficos, animación o sonido. Es por eso que se necesita un PC considerablemente posterior en el tiempo a una consola determinada para que éste la emule con eficacia, puesto que la CPU del PC tendrá que hacer el trabajo de todos esos procesadores.

Existen emuladores de consola, ordenadores, máquinas recreativas, calculadoras, componentes hardware e incluso de máquinas *hand-held* de cristal líquido

En cuanto al software, lo encontraremos en forma de ROMS, antes mencionadas. La idea es que, volcando la información de un cartucho en un archivo, nuestro emulador sea capaz de leer ese software sin hacerle ningún cambio, o en todo caso, añadiéndole sólo algún tipo de cabecera con información útil para nuestro emulador. La emulación de "inserción de un cartucho" podrá realizarse de dos maneras diferentes, dependiendo del tipo de emulador: si se trata de un emulador por línea de comando (MS-DOS) escribiremos el nombre del emulador seguido del nombre de la ROM, con algunos comandos opcionales. En el caso de los emuladores con entorno gráfico, podremos navegar por el sistema de directorios de nuestro disco duro en busca de archivos ROM, y para ejecutarlos bastará

con hacer clic sobre ellos. Este segundo tipo de emuladores comporta, entonces, una dificultad añadida en su programación.

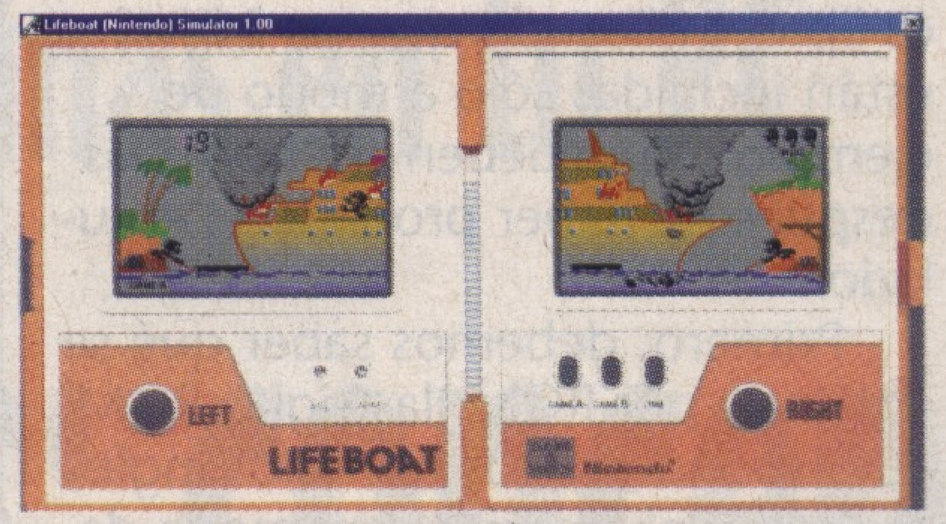
Actualmente, existen emuladores de todas las consolas, excepto de la Dreamcast, cuya tecnología de 128 bit, resulta todavía muy compleja para ser emulada por un PC con eficacia. No sería imposible realizarlo, pero, como es de esperar, los resultados dejarían mucho que desear. He de decir que mis experiencias con emuladores de Nintendo 64 o PlayStation tampoco han sido muy satisfactorias, sin embargo, con consolas menos potentes (de dieciséis u ocho bits) los resultados son auténticamente perfectos, manteniendo la jugabilidad y suavidad de las máquinas originales.

Emuladores de máquina recreativa

Estos son los emuladores con más éxito, ya que son los que permiten tener acceso a los juegos que mayor fama se han granjeado a lo largo de los años. Asimismo, son posiblemente lo emuladores más agradecidos de programar, ya que nos aseguramos que un gran número de gente estará interesada en ellos.

Su funcionamiento es muy similar a los emuladores de consola: el software también se encuentra en forma de ROMS, e incluso la emulación a bajo nivel se basa en la mayoría de los casos en los mismos procesadores que sustentan las consolas de uso doméstico. Un factor nuevo que introducen es el problema del tamaño de pantalla: las máquinas recreativas disponían de monitores muy grandes, que daban cabida a un mayor número de píxels (lo que no quiere decir que tuvieran mayor resolución, puesto que la mayoría de las coin-op trabajaban con una resolución de peso igual a la del modo de inferior calidad de los PCs actuales, es decir, 300x200), lo que nos obliga a visualizar los juegos con una mayor resolución en nuestro monitor, para dar cabida a todos los píxels, con la consiguiente reducción de velocidad y limpieza, y condicionando que se necesiten máquinas más potentes para realizar estas emulaciones. Algunos programadores han optado incluso por una visualización apaisada para aquellas máquinas que tenían una pantalla mucho más alta que ancha.

Podremos enfocar la programación de nuestro emulador como un compilador estático o dinámico, o bien como un intérprete



En plena fiebre de la emulación ninguna máquina se libra (Game&Watch de Nintendo).

Emuladores de ordenador

En este tipo de emuladores, la potencia de la máquina original no representa un problema para la CPU de un PC actual, ya que el avance tecnológico en este campo es vertiginoso y una máquina que fue creada hace 10 años es emulada sin ningún tipo de problema. Incluso la emulación de AMIGA es óptima, pese a que este tipo de ordenadores disponía de un set de coprocesadores gráficos y sonoros similar al de las consolas, antes mencionado.

La obtención y ejecución del software de estos emuladores puede ser más o menos compleja dependiendo del sistema emulado. En el caso de las máquinas cuyo soporte físico es idéntico al de los PCs (disquetes de 3 1/2) , bastará con preparar nuestro emulador para que sea capaz de descifrar formatos diferentes de FAT, y podremos trabajar directamente con los disquetes originales. El sistema AMIGA vuelve a suponer, en este caso, una excepción ya que, aunque funcionaba con disquetes de 3 pulgadas y media y doble densidad que deberían ser leídos sin problemas por la disquetera del PC, el formato resulta físicamente ininteligible para el PC, puesto que el AMIGA aprovechaba una capacidad de 880 k en este tipo de floppys, frente a los 720k que soporta el PC (en discos de doble densidad). Esto provoca que deban crearse imágenes de los disquetes originales del AMIGA en archivos (este proceso debería realizarse en un AMIGA, que sí era capaz de leer los discos de PC), para posteriormente leerlas con nuestro emulador. Por ello, en este y el resto de los emuladores que imitan máquinas cuyo soporte físico era diferente al PC, para poder introducir "discos virtuales" deberemos pulsar una determinada *hot key* (tecla caliente) que nos permitirá elegir uno de estos archivos de imagen para engañar a la ROM original del ordenador emulado, de forma que crea que hay un disquete metido en su unidad y podamos hacer las operaciones correspondientes con él. Tenemos que tener en cuenta que además de los disquetes de 3 1/2, las máquinas antiguas han funcionado con disquetes de 5 1/4 (PCs antiguos), discos de 3 pulgadas (Amstrad CPC y Spectrum +3) con doble cara física (había que darles la vuelta) e incluso dispositivos de cinta (Sinclair, MSX, Commodore64, etc.).

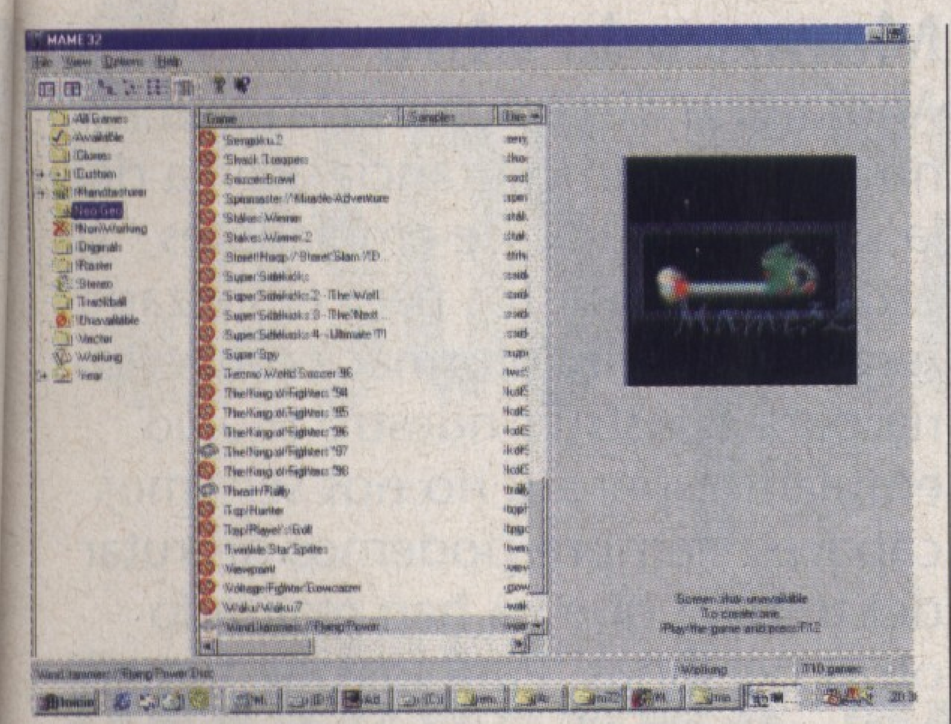
En el caso de los emuladores de ordenadores, las ROMs tienen un cometido diferente, como ya hemos comentado antes. En este caso la ROM será única y será un volcado de la que internamente lleva cada ordenador. Aunque su función sea distinta tienen algo en común con las ROMs de las consolas: su distribución es ilegal. Es por esto que en algunas de las páginas web dedicadas a este tipo de emuladores no se nos provee de estas ROMs, que por cierto, en el caso de la PlayStation también son necesarias, existiendo varias que podremos utilizar según nos convenga, además de las otras ROMs (las que portan la información de los correspondientes juegos).

Otros emuladores

A parte de los emuladores mencionados podremos también encontrar emuladores de calculadoras, por ejemplo. Nos referimos a las tan extendidas super-calculadoras científicas de Hewlett Packard (HP 48), o a otras no tan conocidas como las de Texas Instruments. También hay emuladores de las antiguas máquinas conocidas como Game&Watch, las primeras *hand-held*, precursoras de la omnipresente Gameboy, o de las menos conocidas Game Gear o Atari Lynx. En la página de Emulatronia (*www.emulatronia.com)* podréis encontrar algunos de estos simpáticos juegos.

Seleccionar el lenguaje de programación

Una vez que hayamos decidido el tipo de máquina que deseamos emular tenemos que enfrentarnos a una nueva decisión: ¿qué lenguaje de programación vamos a utilizar? La verdad es que no tenemos muchas alternativas. Para nuestro propósito necesitamos un lenguaje que sea a la vez rápido y potente. El lenguaje que mejor se ajusta a esa definición es el Ensamblador, ya que al ser un lenguaje de bajo nivel permite realizar cualquier tipo de operación, sin límite alguno, y el código



Mame es el más significativo emulador multiplataforma.

generado, al estar pensado en el lenguaje de la máquina, es el más rápido que podremos lograr, pero, como todos sabemos, es un lenguaje muy complejo, que requiere de muchas líneas de código para realizar cualquier operación básica. Muchas de estas operaciones básicas ya están programadas en la mayoría de los lenguajes de alto nivel, pero éstos son más lentos, así que deberíamos elegir un lenguaje de alto nivel que genere código ensamblador lo más rápido posible. Con todo esto, el lenguaje más apropiado es el C, cuya velocidad es la mayor de entre los lenguajes de alto nivel. Aún así, es posible que no obtengamos en algunas partes críticas la velocidad deseada. La solución es codificar estas partes señaladas en Ensamblador.

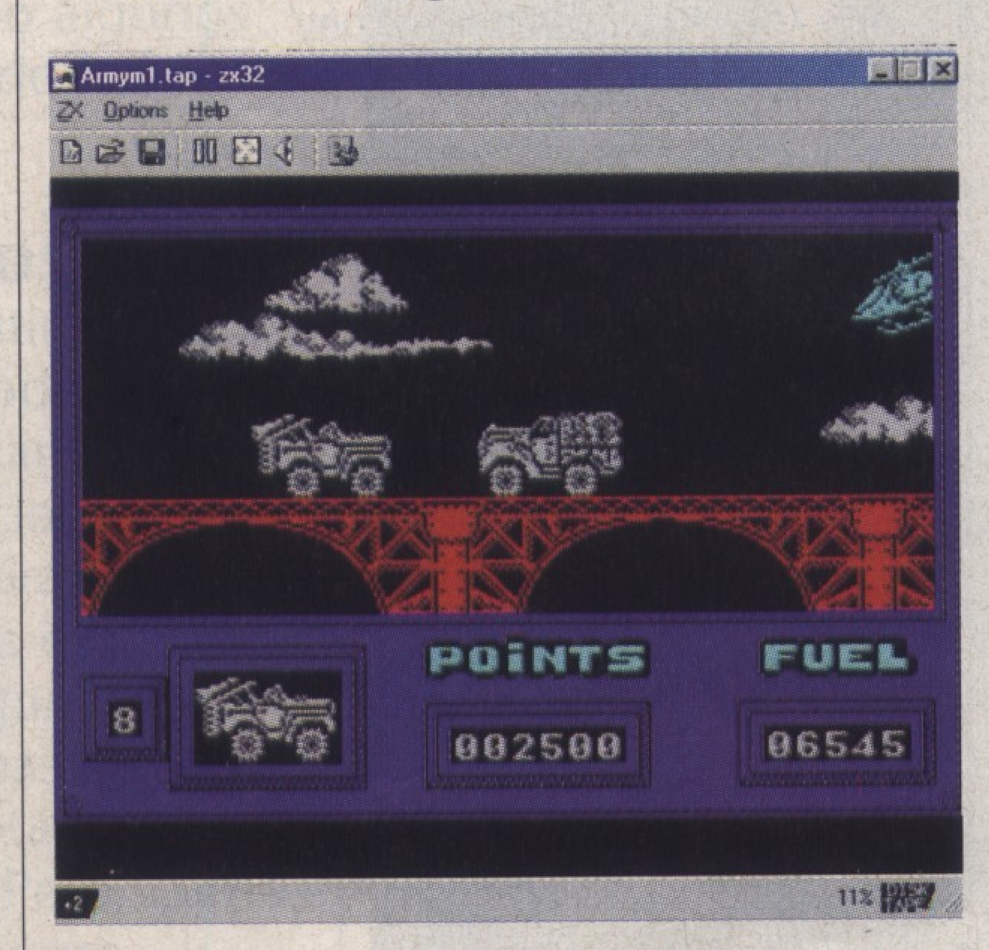
También DIV puede hacerlo

Sí, habéis leído bien. Ha llegado hasta nuestros oídos que alguien había programado un emulador en DIV. No hemos descansado hasta obtener más detalles de esta noticia. Parece ser que alguien que se hace llamar JoseK, ha realizado el emulador de una antigua máquina recreativa llamada Centipede. Esta máquina estaba basada en el procesador 6502 y, según cuenta el propio autor en un archivo LEEME que acompaña al programa en cuestión, ha utilizado un *engine* de dicho procesador programado por alguien llamado Ivan Macintosh y lo ha transcrito al lenguaje de DIV, aprovechando las capacidades de nuestro lenguaje de programación de videojuegos para implementar el aspecto gráfico del Centipede. Toda una obra de arte. Y por si esto fuera poco, hemos podido leer en la página web de JoseK, que también está trabajando en un proyecto bastante más ambicioso: un emulador múltiple para Nintendo Entertainment System. Esperamos ver ansiosos los resultados de su trabajo.

Formas de programar un emulador

Ahora que ya tenemos seleccionado el lenguaje de programación que vamos a utilizar tenemos que decidir de qué manera va a realizar la emulación nuestro programa. Hay tres maneras bien diferenciadas: mediante interpretación, mediante compilación estática o mediante compilación dinámica. Vemos cada una de ellas:

- *Emuladores por interpretación:* Este concepto designa a los emuladores que van leyendo líneas de código original y lo van traduciendo en tiempo real, para irlo ejecutando en nuestro PC. Esta es la forma menos eficiente de emulación, ya que se ocupa gran parte de los recursos de la CPU en la traducción del código, y esto hace que se ralenticen el resto de los procesos. Por eso, esta forma no es apropiada para emuladores de máquinas con bastante potencia gráfica y sonora (Super Nintendo, MegaDrive, NEO GEO, etc.). Sin embargo, es la forma de emulación más apropiada cuando el código de la máquina emulada es automodificable, ya que permite tomar decisiones y realizar cambios sobre la marcha de la ejecución.
- *Emuladores por compilación estática:* Es la forma inversa a la anteriormente descrita. Se caracteriza por hacer toda la conversión en un proceso previo a la ejecución, llamado compilación. Al estar todos los trabajos de emulación realizados con anterioridad, la CPU puede emplear todos sus recursos en la representación gráfica y sonora, realizando estas operaciones con mayor rapidez y suavidad, que es lo que se desea en el caso de los emuladores de las máquinas más sofisticadas. Lo que se hace exactamente es leer primero por completo las ROMs que se van a ejecutar y convertirlas a un código ensamblador equivalente al de nuestra máquina. El problema que tiene este tipo de emulación es que en caso de aparecer código automodificable, no podríamos realizar estas operaciones. Esto significa que no podríamos realizar un emulador de ordenador con este sistema, ya que no sería posible realizar la conversión entre máquinas.
- *Compilación dinámica:* Este es un método híbrido de los otros dos. Se basa en perder algo de la velocidad del segundo método para poder emular código automodificable. Lo que se hace básicamente es realizar la compilación en bloques, según vamos ejecutando la ROM, en lugar de hacer toda la compilación de una vez.



Hay gran cantidad de emuladores del clásico Spectrum.

En conclusión, utilizaremos interpretación para realizar emuladores sencillos que necesiten mucha versatilidad, compilación estática para emuladores de consola y compilación dinámica para emuladores de ordenador.

> Los *engines* son la parte más importante de nuestro emulador

El proceso

Una vez que ya hemos tomado todas las decisiones más importantes, nos ponemos manos a la obra. Lo primero que tenemos que hacer es documentarnos acerca de las características del hardware de la máquina que vamos a emular. Entre esta información debemos encontrar los distintos chips que componen dicho hardware. El siguiente paso es conseguir una definición detallada de las instrucciones que ejecuta cada uno de estos chips, así como una lista precisa de los registros que poseen. Esto último nos será imprescindible para poder programar los *engines* de dichos procesadores.

Programando *engines*

Un *engine* es un fragmento de código que ejecuta un código equivalente a las instrucciones de un procesador cualquiera haciendo uso de las instrucciones propias de nuestro Pentium. Los *engines* son la parte más importante de los emuladores y, a la vez, su parte de más bajo nivel. Ahora bien, dependiendo de la motivación que cada uno tenga para hacer un emulador, podremos programar estos *engines* nosotros mismos, lo cual puede producirnos una mayor satisfacción, o bien obtenerlos de alguna de las páginas web dedicadas a la emulación.

Esto último tiene una importante ventaja, y es que los *engines* de dominio público ya han sido utilizados en muchos emuladores anteriores, de manera que habrán sido testados de manera concienzuda, lo cual asegura su correcto funcionamiento.

Si tomamos finalmente la opción de programar nosotros mismos nuestros motores, tendremos que tener en cuenta que la codificación de un *engine* es bastante compleja, y que conviene programarlo en ensamblador. Lo que haremos será ir construyendo rutinas en ensamblador del x86 que tomarán los mismos parámetros que las instrucciones originales del procesador a emular, y que producirán el mismo efecto que las originales. Asimismo, asignaremos partes de la memoria de nuestro PC para que hagan las veces de los registros del procesador emulado.

Podemos programar el motor del emulador o tomarlo prestado de los ya probados con otros emuladores

El resto del emulador

Ahora que ya somos capaces de ejecutar las instrucciones de los procesadores de la máquina es el momento de utilizar el resto de la información que tenemos acerca del hardware de la misma. Los problemas que se nos plantean ahora son los siguientes:

– Actualización de la pantalla: la complicación de este proceso irá según la calidad gráfica de la máquina original.
– Tratamiento de sprites: Primordial, sobre todo en la emulación de consolas y máquinas recreativas cuyo punto fuerte es la facilidad para mover los sprites por pantalla de manera rápida y suave. Este será uno de los detalles que más influirá en la jugabilidad de las ROMs emuladas.
– Control de joysticks, gamepads, ratones; teclado, etc.
– Tratamiento del sonido: podremos utilizar sonido MIDI en algunos casos, pero también tendremos que utilizar WAV, irremediablemente. Debemos recordar que el sonido es un punto muy importante, pues contribuye definitivamente en la ambientación y sensación de interactividad, y en ningún caso deberíamos permitir que nuestro programa no emulara la música y efectos sonoros de las ROMs originales.
– Generación de interrupciones en la manera en la que lo hacía la máquina original (para leer un dispositivo de control, o reproducir sonido, por ejemplo).

Emuladores multiplataforma

Lo más normal es que nos trabajemos un emulador para una máquina específica, que ya es un loable esfuerzo, pero los más valientes puede que se atrevan a codificar un emulador que sea capaz de hacer funcionar ROMs correspondientes a máquinas con distintas composiciones hardware. En este caso, estaríamos hablando de un emulador multiplataforma. Actualmente, el emulador con mayúsculas que atiende a esta característica es MAME, capaz de trabajar con ROMs de múltiples máquinas recreativas, incluyendo la NEO GEO.

Manos a la obra

Ahora que disponemos de algunas nociones de importancia acerca de la programación de emuladores, estamos dispuestos para empezar a recavar toda la información necesaria para construir nuestro propio emulador, pero si no nos sentimos capaces siempre podemos disfrutar del trabajo que ya han realizado otras personas. Para los que tomen cualquiera de las dos opciones les aconsejamos que se dirijan a cualquiera de las direcciones web que existen sobre el tema o, para una búsqueda rápida y en castellano, aconsejamos de nuevo la página de *www.emulatronia.com*. Que tengáis una emulación satisfactoria.

Sergio Cánovas

Direcciones relacionadas

Aquí tenéis una larga lista de direcciones relacionadas con los emuladores:

http://www.emuclassics.com/tosbox/
http://www.classicgaming.com/cpe/
http://www.emulation.net/msx/
http://www.codepoet.com/UAE
http://www.bleem.com/
http://www.psemu.net
http://www.virtualgamestation.com/
http://ultrahle.emugaming.com/
http://mtr.dht.dk/
http://www.hu6280.com/
http://www.magicengine.com/
http://www.maelstrom.net/callus/
http://hive.speedhost.com/
http://kcbbs.gen.nz/users/chrish/
http://mame.retrogames.com
http://www.classicgaming.com/mame32
http://www.macmame.org
http://xmame.retrogames.com
http://www.rainemu.com/
http://www.retrocade.com/
http://members.xoom.com/ggoedert/dosuae/index.html
http://www.freiburg.linux.de/~uae/
ftp://users.aol.com/davidells/ApplePC/
http://www.pacifist.fatal-design.com/
http://www.computerbrains.com/ccs64down.htm
www.auto.tuwien.ac.at/~rlieger/Power64/Power64.html
http://www.ardi.com/2
http://www.uni-mainz.de/~bauec002/B2Main.html
http://www.lsi.usp.br/~ricardo/brmsx.htm
http://www.komkon.org/~dekogel/fmsx.html
http://personal.redestb.es/raulgomez/r80.htm
http://www.philosys.de/~kunze/xzx/
http://www.geocities.com/SiliconValley/Bay/9932/
http://www.whimsey.com/z26/z26.html
http://stella.atari.org
http://stellax.8m.com/
http://cas3.zlin.vutbr.cz/~stehlik/a800.htm
http://www.shenleyfields.demon.co.uk/rainbow.html
http://www.dysfunction.demon.co.uk
http://internetter.com/titan/software/index.html
http://emu.simplenet.com/lynx/
http://members.xoom.com/ColEmDos/
http://www.komkon.org/fms/ColEm
http://www.work.de/nocash/gmb.htm
http://emulation.net/gameboy/
http://www.komon.org/fms/VGB/
http://www.enterspace.org/world/massage.htm
http://www.komkon.org/fms/MG/
http://www.komkon.org/~stiles/emulation/sega/
http://www.classicgaming.com/meka
http://www.komkon.org/fms/MG/
http://www.emulnews.com/kml-empire/
http://www.dtmnt.com/
http://www.humboldt1.com/~ognir/dgen-sdl.html
http://home5.swipnet.se/~w-50884/emulator/rage.htm
http://underworld.fortunecity.com/simulator/527/
http://www.illusion-city.com/neo/index.htm
http://fwnes.davesvgc.com/
http://207.5.92.43/
http://www.system16.com/m72.html
http://home5.swipnet.se/~w-50884/emulator/rage.htm
http://www.hubcap.demon.co.uk/sparcade.htm
http://www.system16.com
http://abyss.moving-people.net/s16w32.html
http://electron.et.tudelft.nl/~jdegoede/system16.html

Casos prácticos de DIV2

Resolviendo problemas

Continuamos el artículo donde lo dejamos el mes pasado. Este mes veremos las otras funciones que se quedaron en el tintero. Vamos a ello sin más preámbulos.

Bueno, directamente, no vamos a empezar, podría ser algo confuso. Primero resumiremos lo que vimos el mes pasado, para así comprobar el punto exacto donde nos quedamos y continuar a partir del mismo.

En el último capítulo...

El mes pasado dividimos el artículo en dos partes. La primera de ellas era el entorno, dentro de este apartado estuvimos viendo algunos trucos para el generador de *sprites* y los editores de mapas gráficos y 3D.

El otro apartado era el del lenguaje; viendo primero algunas opciones de compilación para evitar errores. También se vieron las funciones 3D, las de búsqueda de caminos y todas las relacionadas con la escritura y manejo de textos.

Vamos a ver todas las instrucciones para manejo de ficheros

Este mes, continuaremos con el lenguaje, y veremos aquellos grupos de funciones que se quedaron en el tintero. Veamos cuales son estas funciones, así como algunos trucos, como viene siendo habitual.

Rutinas de ficheros

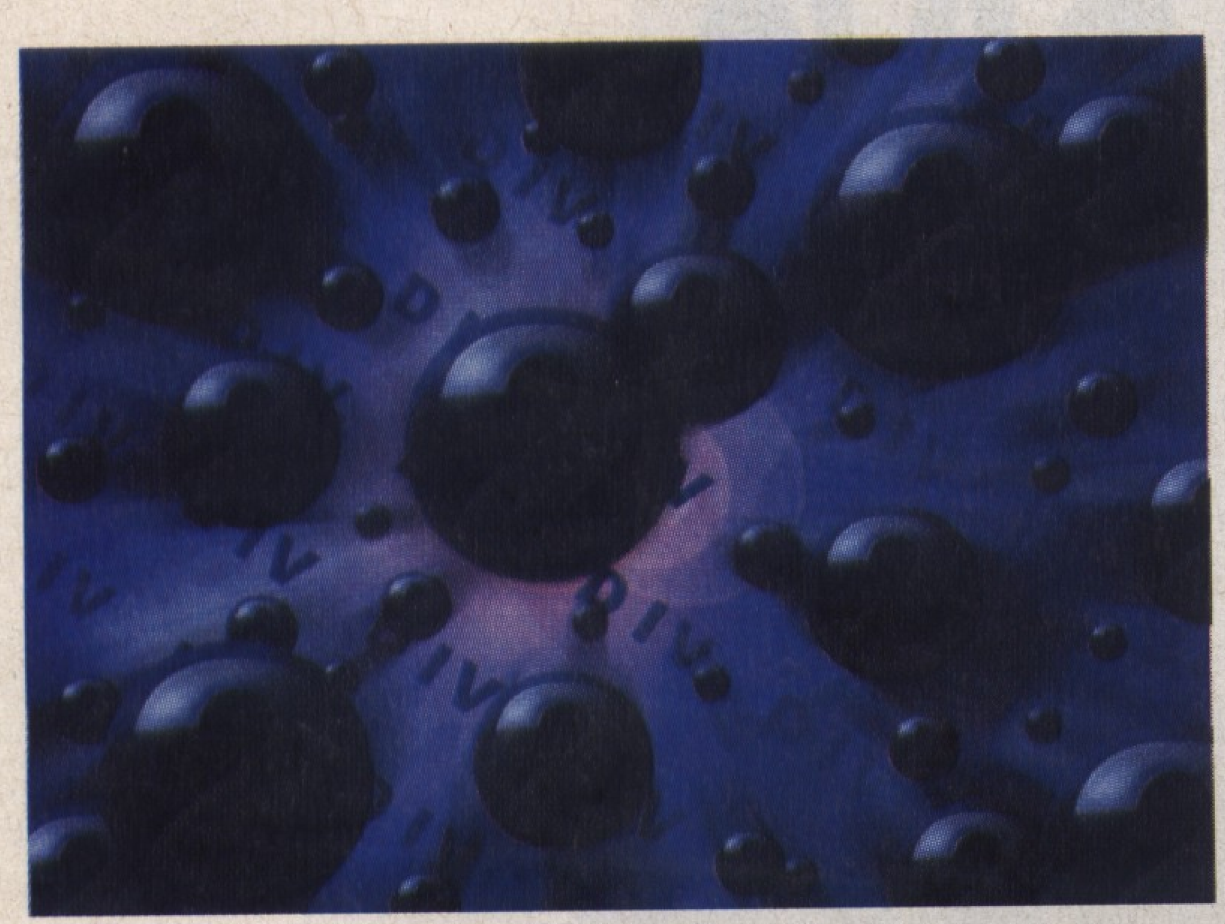
Comenzamos con las rutinas de ficheros, donde se ha dado un gran avance. Antes únicamente se disponía de dos instrucciones para manejo de ficheros. Ahora, la lista de funciones disponibles es bastante extensa, teniendo algunas de ellas similitud con algunas del lenguaje C, siendo otras totalmente novedosas y originales. Pero, como veníamos haciendo, primero comentaremos algunos trucos y, luego, pasaremos a ver la lista de todas estas funciones.

Lo primero a tener en cuenta es el tamaño de lectura y escritura de datos. Este parámetro es modificable, ya que existe una variable global llamada *unit_size,* que determina el número de bytes que se leerán o escribirán cuando se trabaje con ficheros. Este valor es muy importante, por ejemplo, si se quiere leer texto, ya que, en este caso, conviene leer los datos, byte a byte, de uno en uno. Normalmente convendrá leer de esta forma, es decir, unit_size=1, aunque siempre dependerá del tipo de dato que usemos, y del tamaño del mismo. También se puede usar esta variable como un sistema de seguridad, usando un pequeño truco. La idea parte de cuando creamos un fichero y queremos que sólo nosotros podamos crear este tipo de ficheros, para evitar que alguien llegue con un fichero creado por él, lo sustituya, y funcione. En estos casos, a la hora de grabar y leer, se debe definir el valor de *unit_size* con un número primo: siete, nueve, once, etc. Con esto conseguiremos que si el manipulador de ficheros que haga el intercambio no crea el fichero con un múltiplo de bytes igual al parámetro que nosotros hayamos definido, podamos detectar el cambio y actuar en consecuencia.

Y hablando de seguridad en manipulación de ficheros, en este aspecto DIV ha dado un cambio radical, ya desde el comienzo, con la instalación de programas. Con DIV 1, cuando se creaba una instalación de uno de los juegos creados, cualquiera podía modificar los gráficos de los FPG, pudiendo personalizar el juego, cambiando incluso los créditos, si estos venían en un gráfico. Ahora con DIV 2, existe una modalidad de instalación que permite comprimir todos



Con DIV2, es más fácil manejar ficheros de disco.



Poco a poco, iremos desvelando uno a uno todos los trucos de DIV.

los ficheros, así la modificación de los FPG es imposible, ya que, además de comprimirlos, están, hasta cierto punto, codificados con una clave interna de DIV, que nadie, o casi nadie, conoce. Esto está muy bien, ya que a nadie le gusta que se apropien de un trabajo. Pero existen otros tipos de ficheros, siendo unos de los más importantes los que nosotros creamos para guardar datos; tablas de récord, partidas, etc. En estos casos conviene tomar medidas de seguridad y, así, evitar que alguien se ponga el primero en la tabla de récord sin habérselo ganado. Con DIV2, se dispone de algunas funciones para trabajar con ficheros comprimidos y codificados. Con estas funciones podremos comprimir y descomprimir ficheros, con lo que la manipulación al nivel de byte con un editor se dificulta. Pero, siempre, alguien podrá crear una aplicación desde DIV que se encargue de descomprimir ficheros, con lo que se habría salvado esta contrariedad. Por eso, también se dispone de dos rutinas de codificación y descodificación de ficheros. Con el uso de estas funciones la manipulación es imposible, ya que uno de los parámetros que usan es una palabra clave, que modifica la forma de codificar el fichero. Como solamente el programador conoce dicha clave, la modificación es, por tanto, imposible hasta para los mejores crackers.

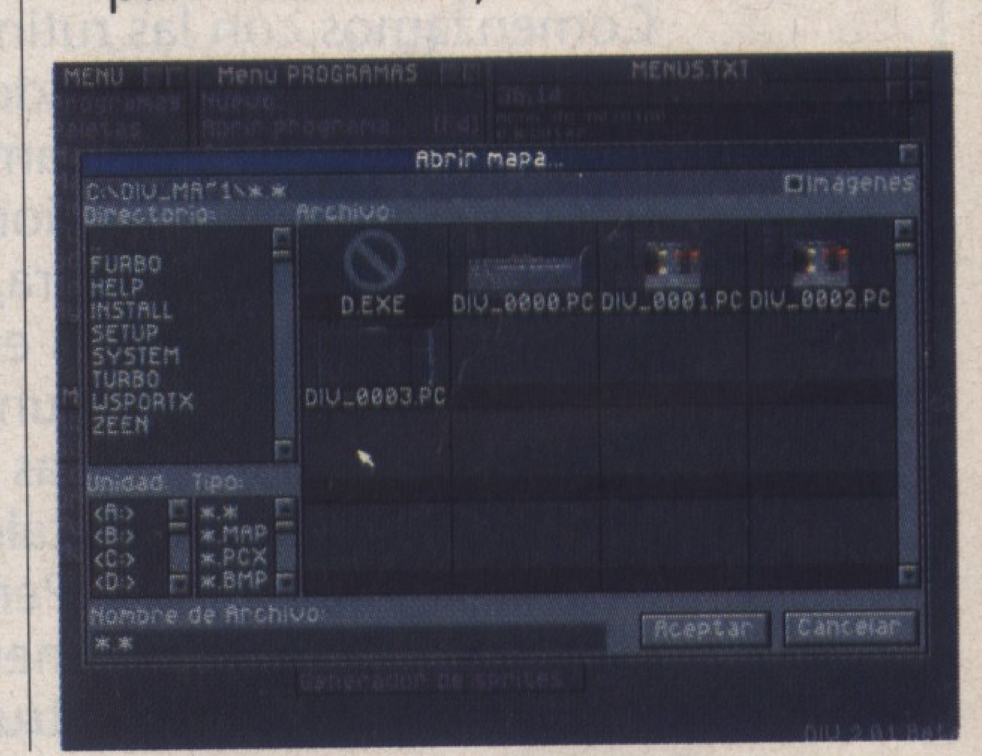
Determinados ficheros se suelen codificar para protegerlos

Algo muy útil es crear lo que se conoce en la programación para Windows como *file manager*. Esto es crear los procesos necesarios, que haciendo uso de las funciones de manejo de directorios, nos permitan movernos por el árbol de directorios del disco duro. En el caso de que se permita al usuario esta opción de navegación, se debe tener en cuenta que, antes de salir del juego, el programador se debe encargar de volver al directorio de inicio donde comenzó, si no lo hace así, el programa dará un error general, que, además de quedar muy feo, podría hasta bloquear el ordenador.

No quiero acabar esta sección de trucos sobre trabajo con ficheros sin comentar un hecho. Se han realizado pruebas con ficheros sin que diesen ningún problema cuando se escriben o se leen de una vez. Es decir, si se abre un fichero, se lee de él o se escribe, luego se cierra y no se vuelve a modificar, no hay problema en ninguno de los casos. Pero también se han hecho pruebas usando dos *handles* de ficheros, en este caso, sí que ha habido problemas. El programa de ejemplo que se usó fue uno cuya aplicación era la de leer de un fichero, modificar datos y guardarlo en otro fichero. Podría ser que el programa de ejemplo no estuviera todo lo depurado posible, sin embargo, la realidad ha sacado a la luz que el fichero destino no guardaba los datos como debiera. Queda en mano de los lectores el uso de más pruebas para comprobar este hecho, aunque es posible que sea un *bug* o fallo de la aplicación.

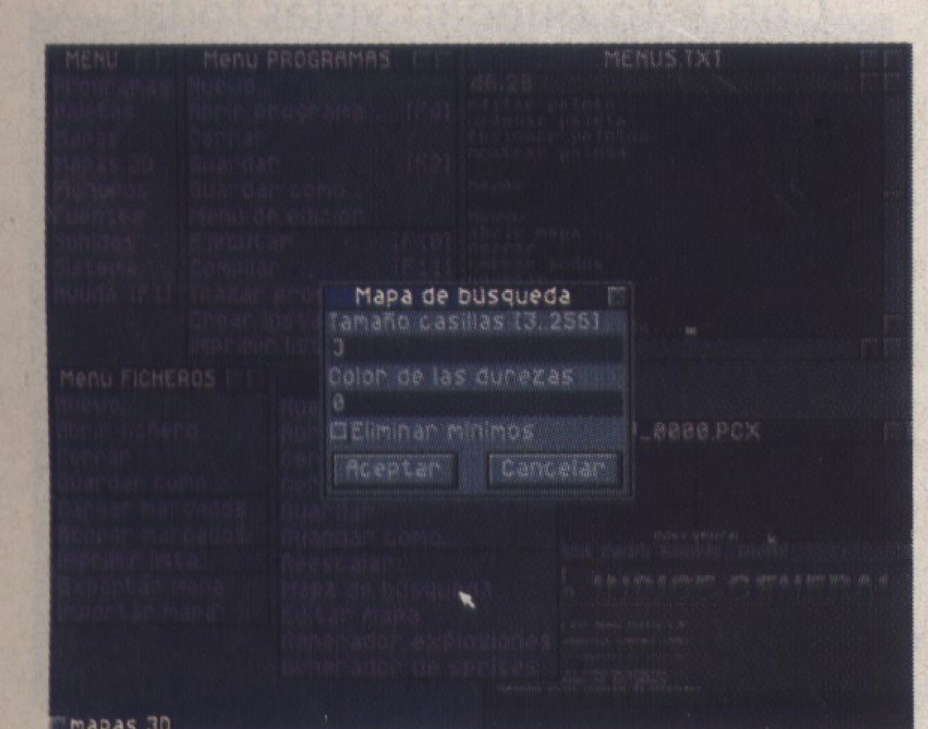
Una vez vistos algunos trucos con ficheros, vamos a ver una lista de las funciones que se usan.

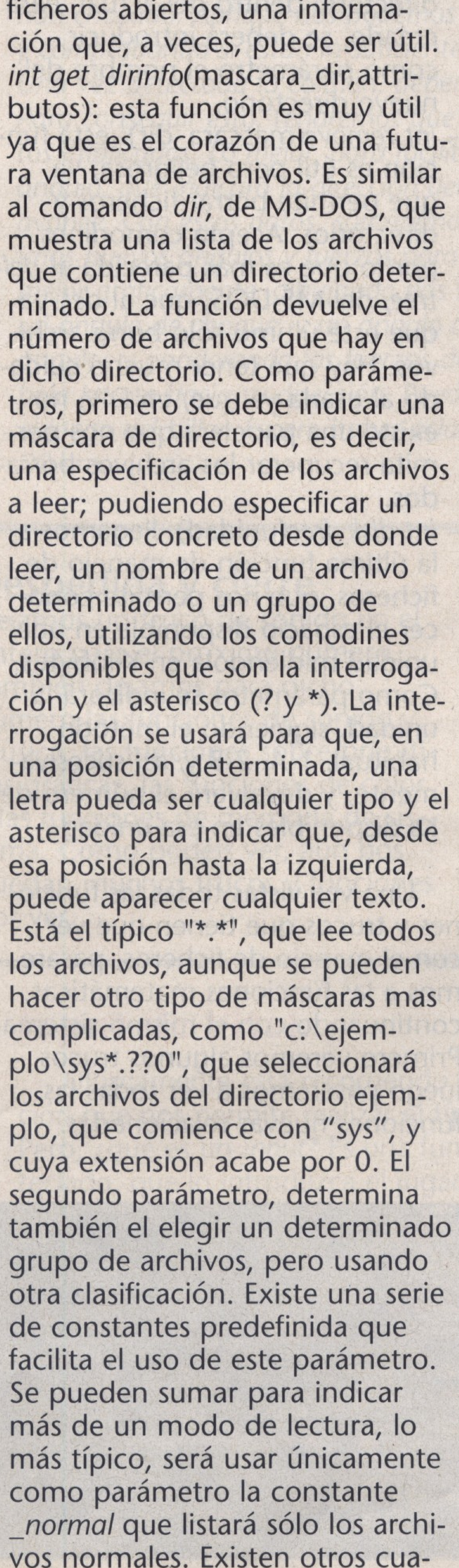
- *int encode_file*(fichero, clave): esta función se encarga de codificar ficheros. Como parámetros, se usan el nombre del fichero dentro del disco duro y una variable del tipo texto, que servirá como clave. Devuelve 1 si ha tenido éxito y 0 si se ha producido un error.
- *int decode_file*(fichero, clave): esta función se encarga de descodificar ficheros. Como parámetros, se usan el nombre del fichero dentro del disco duro y una variable del tipo texto, que servirá como clave, es decir, los mismos parámetros que en el caso anterior. Devuelve 1 si ha tenido éxito y 0 si se ha producido un error.
- *int compress_file*(fichero): con esta función podremos comprimir un fichero en el disco duro. Lo único que se necesita como parámetro es el nombre del fichero a codificar. Devuelve 1 si ha tenido éxito y 0 si se ha producido un error.
- *int uncompress_file*(fichero): esta función realiza el proceso contrario a la anterior. Sirve para descomprimir ficheros. Como en el caso anterior, se necesitará el nombre del fichero a descomprimir. Devuelve 1 si ha tenido éxito y 0 si se ha producido un error.
- *int fopen*(fichero, atributos): ésta es una de las funciones más importantes ya que nos permite abrir ficheros. Como primer parámetro debemos indicar el nombre, con su ruta correspondiente si fuera necesario, del archivo con el que queremos operar. El otro parámetro indica el modo de trabajo con el archivo. Se puede indicar una *r* si se quiere leer del archivo, *w* para escribir o sobrescribir un archivo o usar *a* para añadir datos a un archivo. Esta función es muy parecida a la del lenguaje C, por lo que puede resultar familiar a muchos lectores. La función devuelve lo que se denomina *handle*, una variable especial que sirve para hacer referencia al archivo. Cada vez que queramos hacer algo con un archivo concreto deberemos indicar su *handle* para seleccionarlo, por ejemplo, cuando acabemos de trabajar con dicho archivo, se debe cerrar usando la función *fclose()*.
- *int fclose*(handle): esta función sirve para cerrar un archivo, cada vez que se usa la función *fopen()*, se abre el archivo, después de trabajar con el mismo se debe cerrar. Como parámetros se debe indicar el *handle* del archivo a cerrar, este *handle* fue devuelto por la función *fopen*, que abrió dicho archivo. Esta función también es similar a una que existe en el lenguaje C.
- *int fread*(&buffer,longitud,handle): esta función se usa para leer datos desde un archivo. El parámetro &buffer debe ser la dirección de una variable donde se guardarán los datos. El parámetro longitud indica el numero de datos a leer y la variable *handle* será, como su nombre indica, el *handle* del archivo desde donde se lee. Esta función también tiene su hermana "mayor" en el lenguaje C.
- *int fwrite*(&buffer,longitud,handle): si la anterior función servía para leer datos, ésta en concreto



sirve para escribir, es decir, el proceso contrario. Se necesitarán, como parámetros, una variable desde donde leer, una longitud, que determinará el número de datos a escribir, y el *handle* del archivo donde se vaya a escribir. Como la función anterior, también tiene su idéntica en C. Con las dos funciones, ésta y la anterior, podremos trabajar con archivos, aunque se deben tener en cuenta los atributos usados al abrir el archivo.

- *int fseek*(handle,posición,modo): esta función sirve para colocarse dentro de un archivo. Usa tres parámetros, el primero es el *handle* del archivo, el segundo es el número de bytes que hay que desplazarse dentro del archivo, y por último, tenemos el modo, que indica desde donde se debe desplazar dentro del archivo. Existen unas constantes definidas que facilitan el uso de este parámetro. Si se utiliza como modo, el valor *seek_set*, se usará como inicio de desplazamiento el inicio del archivo. Si se desea desplazarse desde el final del archivo se usará *seek_end* y, por último, se utilizará *seek_cur* para desplazarse desde la posición donde se encuentre actualmente el fichero abierto, lo que se podría denominar "desde la posición del cursor de desplazamiento".
- *int ftell*(handle): si se desea saber la posición de lo que denominamos cursor de desplazamiento, que indica el lugar del archivo donde se leerán o escribirán datos, se podrá hacer usando esta función, que devuelve dicha posición. Este valor devuelto no tiene nada que ver con el modo de la anterior función, ya que el parámetro de esta función *ftell()* indica la posición dentro del archivo y el anterior parámetro indicaba un modo de desplazamiento (inicio, final o cursor).
- *int filelength*(handle): con esta función podremos conocer la longitud de un fichero. Es decir, el número de datos que contiene. Como parámetro necesita el *handle* de un fichero abierto.





Cuando se usa esta función no se modifica la posición de lectura donde esté el archivo. El valor que devuelve, como se ha dicho, es el número de datos que contiene ese archivo.

- *int flush*(): esta función tiene dos cometidos. Uno de ellos consiste en vaciar los buffer de escritura y no es de mucha utilidad. Cada vez que se escribe en un archivo los datos pasan por un buffer, esta función consigue que se escriban automáticamente todos los datos que halla en dichos buffer, vaciándolos. Esto, normalmente, no es necesario, ya que el sistema tiene una gran optimización en estas operaciones y suele ser invisible para el programador. El otro cometido de la función tiene que ver con el valor devuelto, ya que indica el numero de ficheros abiertos, una información que, a veces, puede ser útil.
- *int get_dirinfo*(mascara_dir,atributos): esta función es muy útil ya que es el corazón de una futura ventana de archivos. Es similar al comando *dir*, de MS-DOS, que muestra una lista de los archivos que contiene un directorio determinado. La función devuelve el número de archivos que hay en dicho directorio. Como parámetros, primero se debe indicar una máscara de directorio, es decir, una especificación de los archivos a leer; pudiendo especificar un directorio concreto desde donde leer, un nombre de un archivo determinado o un grupo de ellos, utilizando los comodines disponibles que son la interrogación y el asterisco (? y *). La interrogación se usará para que, en una posición determinada, una letra pueda ser cualquier tipo y el asterisco para indicar que, desde esa posición hasta la izquierda, puede aparecer cualquier texto. Está el típico "*.*", que lee todos los archivos, aunque se pueden hacer otro tipo de máscaras mas complicadas, como "c:\ejemplo\sys*.??0", que seleccionará los archivos del directorio ejemplo, que comience con "sys", y cuya extensión acabe por 0. El segundo parámetro, determina también el elegir un determinado grupo de archivos, pero usando otra clasificación. Existe una serie de constantes predefinida que facilita el uso de este parámetro. Se pueden sumar para indicar más de un modo de lectura, lo más típico, será usar únicamente como parámetro la constante *_normal* que listará sólo los archivos normales. Existen otros cuatro tipos que son: *_hidden* para incluir los archivos ocultos, *_system* que incluirá los archivos del sistema, *_subdir* para incluir los subdirectorios y por último *_volid* para listar sólo la etiqueta del volumen. Una vez determinado los parámetros y lo que retorna la función, veremos cómo utilizarla. Cuando se llama a esta función, se rellena una estructura llamada *dirinfo*, que está definida de la siguiente manera:



El manejo de textos, es superior en esta nueva versión.

```
STRUCT dirinfo;
files;
name[1024]; END
```

En el campo files se guardará el numero de archivos en lista, que será igual al devuelto por la función *get_dirinfo()*. Y en la tabla *name*, habrá una lista de punteros a los nombres del directorio. Es decir, cuando se llame *get_dirinfo()*,el número de archivos que cumplían las máscaras y modos de lectura se guardarán en esta estructura, dentro de *name[]*, cada uno de los nombres de los archivos. En *dirinfo.name[0]*, se encontrará el nombre del primer archivo, en *dirinfo.name[1]*, el siguiente y, así sucesivamente. Estos archivos se guardan en orden alfabético, algo a tener en cuenta, y puede darse el caso de que ningún archivo cumpla las condiciones introducidas. También es bueno leer los archivos por un lado y los subdirectorios por otro, usando para ello, las constantes *_normal* y *_subdir*, respectivamente.

Hasta hace poco sólo había dos instrucciones para manejo de ficheros

- *get_fileinfo*(nombre): esta función sirve para coger información sobre un archivo determinado. El nombre de archivo deberá ser indicado como parámetro; la función devolverá 0 si el archivo no existe, o no se ha podido obtener información sobre él, y 1, si la función ha tenido éxito. En

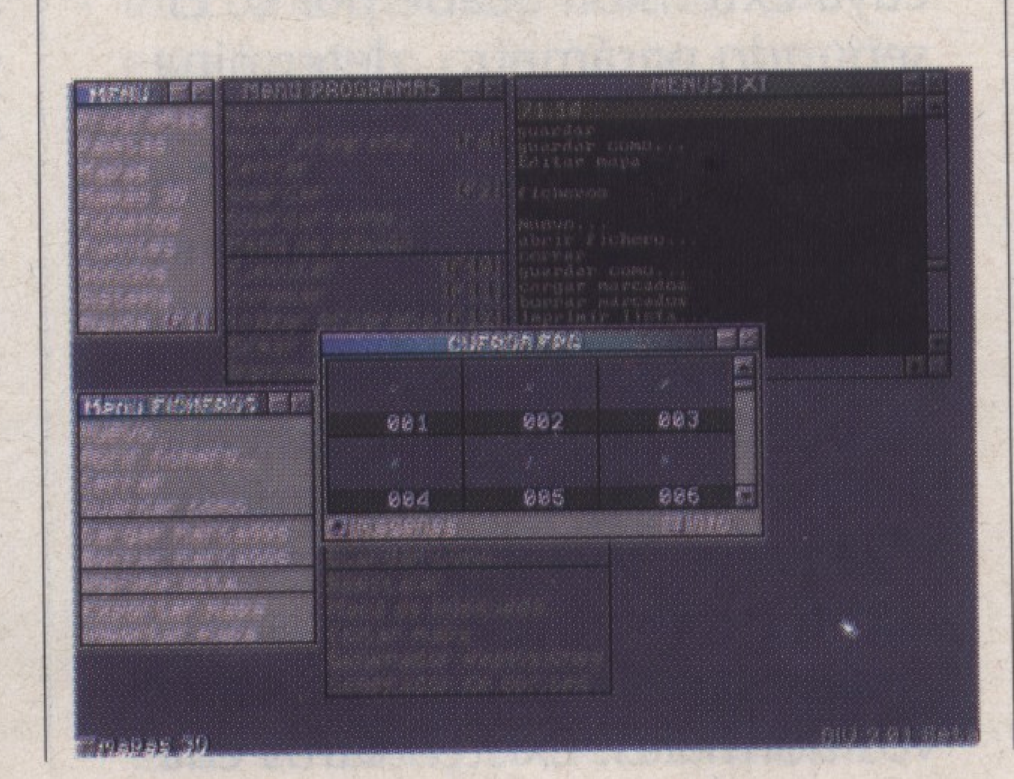

Ahora se pueden hacer juegos de tipo DOOM, gracias a las opciones 3D.

este caso, la información sobre el archivo se guardará en la estructura *fileinfo*, que tiene diversos campos. Por un lado está *fullpath*, que indica el nombre completo, incluyendo la ruta, *drive*, que indicará la unidad de disco, siendo 1-A:, 2-B:, 3-C:, etc. También está *dir*, que guardará el directorio del archivo, *name*, que guardará el nombre y *ext* la extensión del mismo. Luego esta size que indica el tamaño indicándolo en datos. También están *day*, *month* y *year*, que indican el día, el mes y el año, respectivamente, de la fecha de última actualización; y *hour*, *min* y *sec* que indicarán la hora, minuto y segundo de la última actualización. Por último, está *attrib*, que indica los atributos del archivo. Como se puede comprobar, toda esta información es la que sale normalmente cuando se consigue una lista de archivos, por ejemplo con *dir* del MS-DOS, pero, en este caso, sólo se obtendrá la información de un archivo determinado. Lo mejor, para comprobar su funcionamiento, es hacer algunas pruebas y comprobar los campos con la información obtenida.

Las funciones matemáticas son útiles para crear la física de los objetos

- *int getdrive()*: esta función sirve para saber en qué unidad de disco se encuentra actualmente el sistema, ya que devuelve la unidad actual siendo 1-A, 2-B y, así sucesivamente, hasta el número de unidades de que se disponga.
- *int setdrive*(unidad): ahora nos encontramos con, lo que se podría definir, función contraria a la anterior. Ésta se usa para fijar la unidad actual, funcionando de igual manera que la anterior, ya que se debe introducir un número que indique el número de unidad, 1-A, 2-B y, así sucesivamente. Para poder comprobar todas las unidades, se debe ir llamando a *setdrive()*, incrementando la unidad de disco duro, y ver si realmente ha cambiado o no, con *getdrive()*, ya que no se devuelve ningún error. Además, este método es útil para conocer las unidades disponibles.
- *int chdir*(dir): para la gente que conozca los comandos de MS-DOS, esta función sería la más similar al comando *cd*, que nos permite cambiar de directorio. Para ello deberemos pasar el nombre del directorio al que nos queremos mover. Devolverá 1 si lo ha hecho con éxito y 0 si se produjo un error.
- *int mkdir*(dir): si la anterior función era similar al comando cd del MS-DOS, ésta lo es al MD, que permite crear un nuevo directorio dentro del actual. Para crearlo, se deberá introducir como parámetro el nombre del nuevo directorio.
- *int remove*(mascara_dir): esta función es útil pero peligrosa, ya que permite borrar ficheros y directorios. Acepta comodines, por lo que es más parecida al *deltree*, del MS-DOS, que al *del*, ya que este último sólo borra archivos. Hay que tener mucho cuidado al usarla ya que, en DIV, no existe una papelera que nos permita recuperar los archivos borrados.
- *int disk_free*(unidad): llegamos a la última función de manejo de ficheros, que nos permite conocer el espacio disponible en una unidad midiéndola en Kilobytes. Como parámetro se indicará la unidad, siguiendo el método habitual, 1-A, 2-B y, así sucesivamente, y devolverá el número de Ks disponibles en esa unidad.

Una vez vistas todas las funciones y trucos que tienen que ver con el manejo de ficheros, pasaremos a las funciones matemáticas, continuando con el mismo sistema. Primero veremos algunos trucos, luego pasaremos a ver todas las funciones nuevas relacionadas.



Funciones matemáticas

Ahora veremos todas las funciones matemáticas. Éstas pueden parecer de uso específico para matemáticos pero no es así, ya que son muy útiles para crear la física de los objetos, como botes, inercias, etc. Se pueden estudiar fórmulas de física en cualquier libro y, en algunas de ellas, se necesitan ciertas funciones matemáticas que son las que veremos al final de esta parte.

Y para muestra, un botón, de modo que veremos un ejemplo de la utilidad de estas funciones. Supongamos que queremos hacer un círculo, punto a punto, en el fondo de pantalla. Para calcular cada uno de estos puntos, teniendo un ángulo dado, y pasándolo a radianes, usaremos las funciones de seno y coseno para conseguirlo.

Esto también se ve en la similitud de las funciones *sin()* y *cos()*, como las que había anteriormente en DIV, que eran *get_distx()* y *get_disty()*. Siendo las primeras, como de más bajo nivel, ya que en las segundas se incluye un radio de circunferencia y en las primeras no.

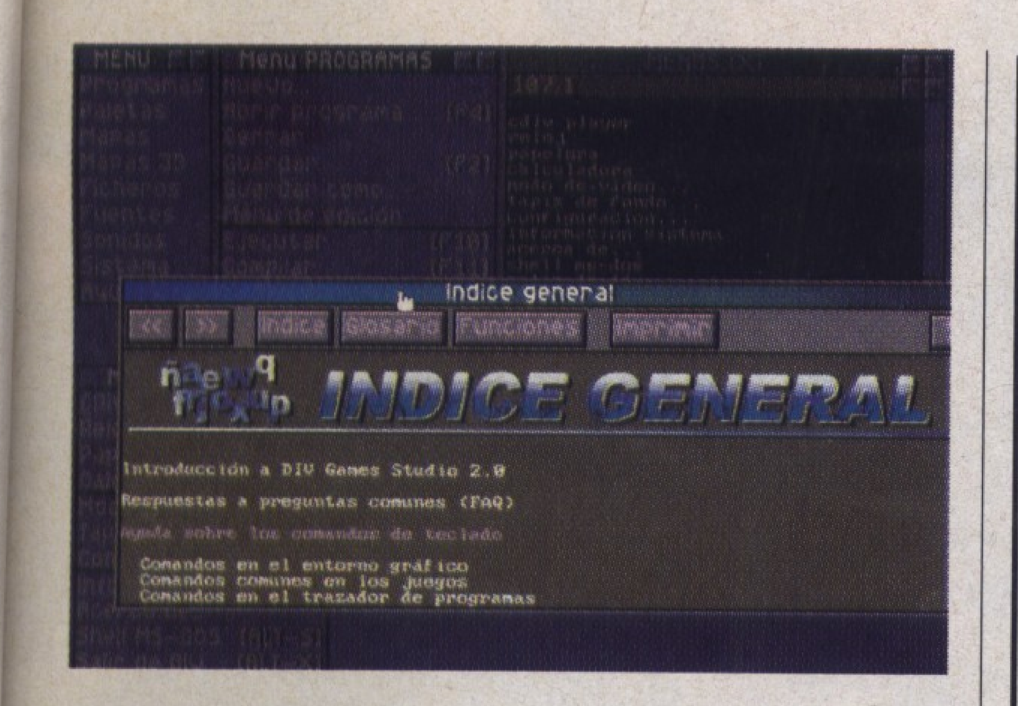
Las otras funciones nos calcularán la tangente, el arco del seno, el arco del coseno y el arco de la tangente utilizando dos métodos distintos. Todos los datos se darán en milésimas, ya que estas funciones, normalmente, trabajan con varios decimales.

- *int sin*(angulo): halla el seno de un ángulo dado como parámetro.
- *int cos*(angulo): halla el coseno de un ángulo dado como parámetro.
- *int tan*(angulo): halla la tangente de un ángulo dado como parámetro.
- *int asin*(seno): halla el ángulo del arco de un seno dado.
- *int acos*(coseno): halla el ángulo del arco de un coseno dado.
- *int atan*(tangente): halla el ángulo del arco de una tangente dada.
- *int atan2*(x,y): halla la arcotangente obtenida de dividir los dos valores pasados como parámetro, que es un ángulo.

Con esto quedan vistas todas las funciones matemáticas, veremos ahora el sonido y la música, un campo muy importante en cualquier vídeojuego, en el que se han introducido muchas mejoras.

El sonido y la música

En esta última sección únicamente veremos los trucos, dejando la lista de función, con las especificaciones propias, para un próximo artículo.



Así que, sin más preámbulos, pasemos a ver dichos trucos para sacar mas partido al sonido en DIV2.

El primer truco consiste en los volúmenes. En la anterior versión de DIV, el volumen se ponía al tope cuando entraba algún juego, teniéndolo que bajar. En esta versión ha pasado lo contrario, está demasiado bajo, a veces, inaudible. Lo mejor es subir todas las variables referentes a los volúmenes de los archivos de sonido del tipo wav y ponerlas al máximo. Otro truco consiste en amplificar dichos archivos wav con programas externos, como el CoolEdit, que dispongan de la opción de amplificación del volumen de sonido.

Cambiamos de tercio ya que ahora comentaremos otro nuevo aspecto: incluir temas de música del formato IT, XM, MOD, S3M. Estos ficheros se realizan con unos programas de creación de música, que se denominan comúnmente *trackers*. Algunos de los más conocidos son el Screen Tracker, el Impulse Tracker o el Modplug, que es para Windows. Todos ellos funcionan de forma parecida, los instrumentos son *samples*, es decir, ficheros con sonidos, parecidos a los wav, pero sin cabecera. El programa tiene varios canales donde podremos ir introduciendo estos sonidos en forma de listas o paternas. Con esto conseguiremos, por ejemplo, si disponemos del sonido de un bombo, una caja, unos charles, un platillo, etc. realizar un ritmo de batería. La mayoría de estos programas son bastante fáciles de manejar y se dispone de muchos instrumentos grabados por otros compositores. En este aspecto, el intercambio de instrumentos es habitual entre los distintos temas.

Consultas de nuestros lectores

Hola, Tizosoft, el motivo de ponerme en contacto contigo es para ver si me puedes dar un concepto generalizado sobre como se debería gestionar en DIV los rebotes de una pelota en un entorno en modo 7, o bien, en modo 8 contra el suelo, o bien, contra otros procesos, ya que estoy planteándome el realizar en mis ratos libres un juego de fútbol, gracias y un saludo.

Fernando Nicolás Torres.

Amigo Fernando, espero que no te moleste que incluya esta pregunta en la revista pero me ha parecido interesante. En un juego de fútbol, existen varios tipos de rebotes. Haremos dos grupos, uno contra el suelo, el rebote del balón, y el otro con otros elementos, como pueden ser los jugadores o los palos de la portería. Para el primero, para que el bote sea real, debes conseguir que el balón se comporte como si estuviera dentro de un circulo, por lo menos en el eje en el que bota. Es decir, usando las funciones get_disty() o, si la prefieres, la función de coseno. Deberás hacer que el balón suba y baje, pero sólo media circunferencia, la otra está, por decirlo de alguna manera, bajo tierra. En cuanto al rebote con los otros elementos de juego, el primer problema con el que nos encontramos es que no existe la función collision(), para detectar si han chocado. Aunque nos podemos crear una usando las coordenadas y comprobando si están lo suficientemente cerca como para colisionar. Una vez detectada la colisión, deberemos hacer que choquen y reboten. Para conseguir esto, tendremos que usar algunos datos de los elementos implicados. Lo mejor, para facilitar los cálculos, es trabajar con un ángulo, que indicará la dirección del objeto, y una velocidad o fuerza de avance. Con estos datos, sumaremos los ángulos y fuerzas, resultando unos nuevos ángulos y fuerzas para los elementos implicados. Esto se ve mejor gráficamente, dibujando dos bolas de billar que chocan, tendrán un ángulo de choque, y un ángulo de trayectoria, sumando estos dos ángulos y ponderando las fuerzas de las bolas, conseguiremos que cada bola rebote hacia su dirección. Puedes ver un ejemplo dentro de Div, en el juego del billar. Espero haber aclarado tu duda y que sirva para otros DIVeros.

Despedida y cierre

Dejamos un truco de sonido en el tintero para el próximo numero. También dejamos para dicho número las funciones de sonido y música, las funciones sobre primitivas gráficas, las funciones de color y las funciones de memoria de sistema. Como podéis ver, aún nos quedan muchos trucos y funciones por ver.

Espero que os sean útiles los trucos que hemos visto este mes, continuaremos con la tarea el mes que viene. Podéis mandar vuestras dudas al e-mail: tizo@100mbps.es, o a la dirección postal de la revista. Espero que el mes que viene volváis a leerme y que, mientras, os DIVirtáis programando.

Tizosoft (Tizo@100mbps.es)

Input de Vital...

Últimamente está recorriendo la Red, un programita que contiene una función que nos permite realizar el típico *Input* de otros lenguajes, como el Basic. Pero, ¿qué es un *input*? Es una función que permite recoger un dato desde el teclado, que lo introduzca el jugador y guardarlo en una variable. Como DIV, estaba realizado especialmente para jugar, disponía de funciones de lectura de teclado pero orientada más a la lectura rápida de una tecla, por ejemplo, leer la tecla del cursor izquierdo para comprobar si el jugador quiere moverse hacia esa dirección. Pero no estaba preparado para, por ejemplo, recoger el nombre del jugador en la tabla de récord. Con el Input de Vital esto es posible, ya que funciona perfectamente, permite repetición de letras, borra bien los caracteres y lee casi todo el teclado, pudiendo utilizar símbolos, mayúsculas, minúsculas, etc. "Un aplauso pa'l Vital", hasta se podría hacer un procesador de textos con él, pero eso es otra historia que se contará en su momento.

Arcade/Plataformas

Toma de contacto

La programación básica de un juego de arcades o plataformas no requiere un nivel avanzado de conocimientos pero sí grandes dotes de imaginación. En este primer artículo trataremos el movimiento del personaje y su interactividad con el entorno o paisaje.

Tanto la programación de arcades como de plataforma disfrutan en el sector de la programación lúdica de una ventaja considerable: en este terreno las técnicas están claramente definidas. Una ventaja que no impide mejoras o innovaciones en el mismo pero deja sentadas unas bases fundamentales que nos permitirán avanzar en su programación de forma rápida. No adelantemos acontecimientos, este primer artículo lo dedicaremos a la toma de contacto relacionado con el arcade y las plataformas de forma práctica, dando un repaso a los movimientos de los personajes en pantalla y la interacción con el entorno, en concreto el mapa de durezas.

La programación de este tipo de juegos tiene unas técnicas claramente definidas

La interacción con el entorno

La interacción con todo lo que rodea a nuestros personajes es una parte esencial en el juego, que va a determinar su grado de realismo.



La combinación de arcades y plataformas, claves del éxito.

En la medida de lo posible, el proceso que programemos para la relación de los personajes con el entorno debe resultar transparente para el jugador, es decir, crear un sistema físico natural, una adaptación a los contornos real y una coordinación eficaz.

Precisamente porque nuestro entorno no va a ser estrictamente geométrico debemos ayudarnos de los mapas de durezas. Pero ¿qué es exactamente un mapa de durezas? El mapa de durezas es un mapa que tendremos siempre en memoria, no se presentará en pantalla, y que utilizaremos de forma paralela al que está viendo el usuario. Este contendrá una serie de datos que, trasladados al mapa real del juego, nos permitirá saber si nuestro personaje está chocando contra una pared, está situado en una plataforma, si ha caído a un barranco o hacer que se adapte al contorno de una colina.

Programando el mapa de durezas

El mapa de durezas se suele realizar sobre una copia, escalada a la mitad del tamaño real, del fondo normal del juego y en escala de grises. Este sencillo paso se puede realizar incluso dentro del entorno de DIV en *[Menú mapa, reescalar]* señalando las opciones de *proporcional* y *escala de grises*. Una vez creada esta base, trabajaremos sobre ella repasando con líneas de colores los contornos del terreno y allá por donde se pueda pasar. Hay que tener en cuenta tres factores: utilizar, siempre que sea posible,



El punto de control 0.

los mismos colores para todos los mapas de durezas para poder aprovechar el código; usar colores diferentes para cada tipo de región; y la más importante, aumentar el grosor de las líneas de color en las secciones del terreno que sean especialmente accidentadas. Éste último aspecto ha de tenerse muy en cuenta debido a que los métodos que se utilizan para detectar el contorno en un mapa de durezas no son totalmente perfectos y tienden a crear efectos no deseados en este tipo de situaciones.

Empecemos por la programación del personaje y avancemos hasta el mapa de durezas. Debemos saber que el movimiento del personaje no va a ser de la misma forma que en cualquier otro juego, es decir, con un simple bucle que controle con una variable global la dirección y, según se toquen las teclas de dirección, ponemos las animaciones correspondientes. En este caso, debemos tener en cuenta que para mover el personaje debemos: recibir información de lo que el jugador desea





hacer, consultar con el entorno si el movimiento es posible, las consecuencias que va a traer y devolver el resultado, como decíamos, de forma transparente para el jugador.

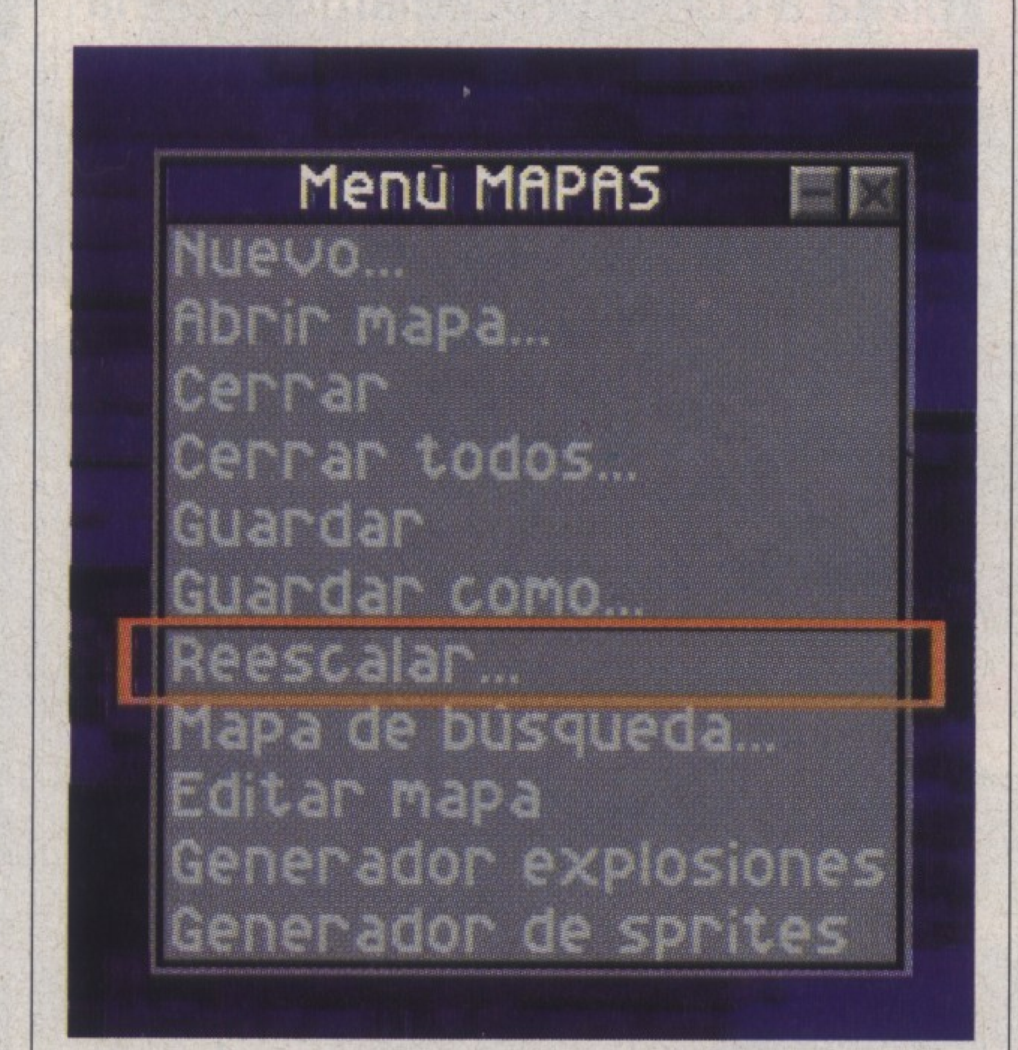
El método general que se usa para el control del movimiento del personaje cuando se usan los mapeados de durezas para este tipo de juegos (plataformas 2D en vista lateral) se suele enfocar basándose en el efecto de gravedad. El efecto de gravedad lo vamos a definir como un refrán que aplicaremos a la programación "si no está en el suelo ha de caer hasta que lo encuentre con velocidad en aceleración constante". De esta manera, el bucle del movimiento del personaje va a empezar comprobando si está en el suelo y, si en algún momento no lo estuviera, hará que el personaje caiga en aceleración constante. Para ello en DIV utilizaremos *map_get_pixel(<fichero>,<grafico>,<x>,<y>)* que nos permitirá consultar el color específico de un mapa que está en pantalla y no en memoria (recuérdese que el que consultaremos no es el mapa de pantalla, sino el mapa de durezas). Note igualmente que el mapa está normalmente escalado a la mitad de su tamaño original, por lo que el código para controlar si el personaje está en el suelo sería el siguiente: *IF(map_get_pixel(fichero,grafico,x/2,y/2)<>color_a_consultar).*

En caso de ser diferente al color_a_consultar, que sería el color que le asignamos en mapa de durezas al suelo, el personaje debería caer con aceleración constante hasta encontrarlo, lo que en principio se reduciría a:

```
WHILE (map_get_pixel(fichero,
grafico,x/2,y/2)<>color_a_consultar)
      gravedad+=2; IF (gravedad>
20)gravedad=20;END //un simple
limite
      y-=gravedad;
END
```

No obstante, debemos tener en cuenta que si la gravedad fuera mayor a un píxel podríamos sobrepasar la línea del suelo y perder al personaje sin remedio por la parte inferior de la pantalla. Debemos incluirle un bucle que compruebe uno a uno los píxeles de esa caída y parar el bucle en el momento en que se encuentre con el suelo:

```
WHILE(map_get_pixel(fichero,gra
fico,x/2,y/2)<>color_a_consultar)
   gravedad+=2;
   //ponemos un límite máximo a
la gravedad
   IF(gravedad>20)
      gravedad=20;
   END
   FOR(contador=0;contador
<gravedad;contador++);
      IF(map_get_pixel(fichero,
grafico,x/2,(y+contador)/2)==color a
consultar)
         gravedad=0; //para que
no siga bajando
         y+=contador; //aplicamos
el incremento exacto
         BREAK; //y cortamos el
bucle
      END
   END
END
//y si el contador se ha igualado a la gravedad es que no hemos
encontrado en ningún momento el
suelo, por lo que damos vía libre a la
gravedad
   IF (contador==gravedad)
y+=gravedad; END
```



Con DIV reescalar y transformar a escala de grises map's es tarea fácil.

Con este paso ya hemos resuelto algunas cosas de nuestro mapa de durezas. En principio, hemos hecho que cuando nuestro personaje caiga de una plataforma descienda aceleradamente hasta encontrar el suelo pero, además, hará que el personaje se adapte a la parte descendente de una ladera, puesto que al avanzar recto el personaje no estará pegado al suelo y el efecto de gravedad se encargará de ponerlo en su sitio. Sin embargo, aún nos quedan grandes partes del código como el control de las teclas del jugador, que en este caso se reducen a izquierda y derecha. Al igual que ocurre con la gravedad, el movimiento del personaje a izquierda y derecha debe ir precedido de una serie de instrucciones que controlen que no se salga de lo trazado en el mapa de durezas. Veamos cómo sería con código:

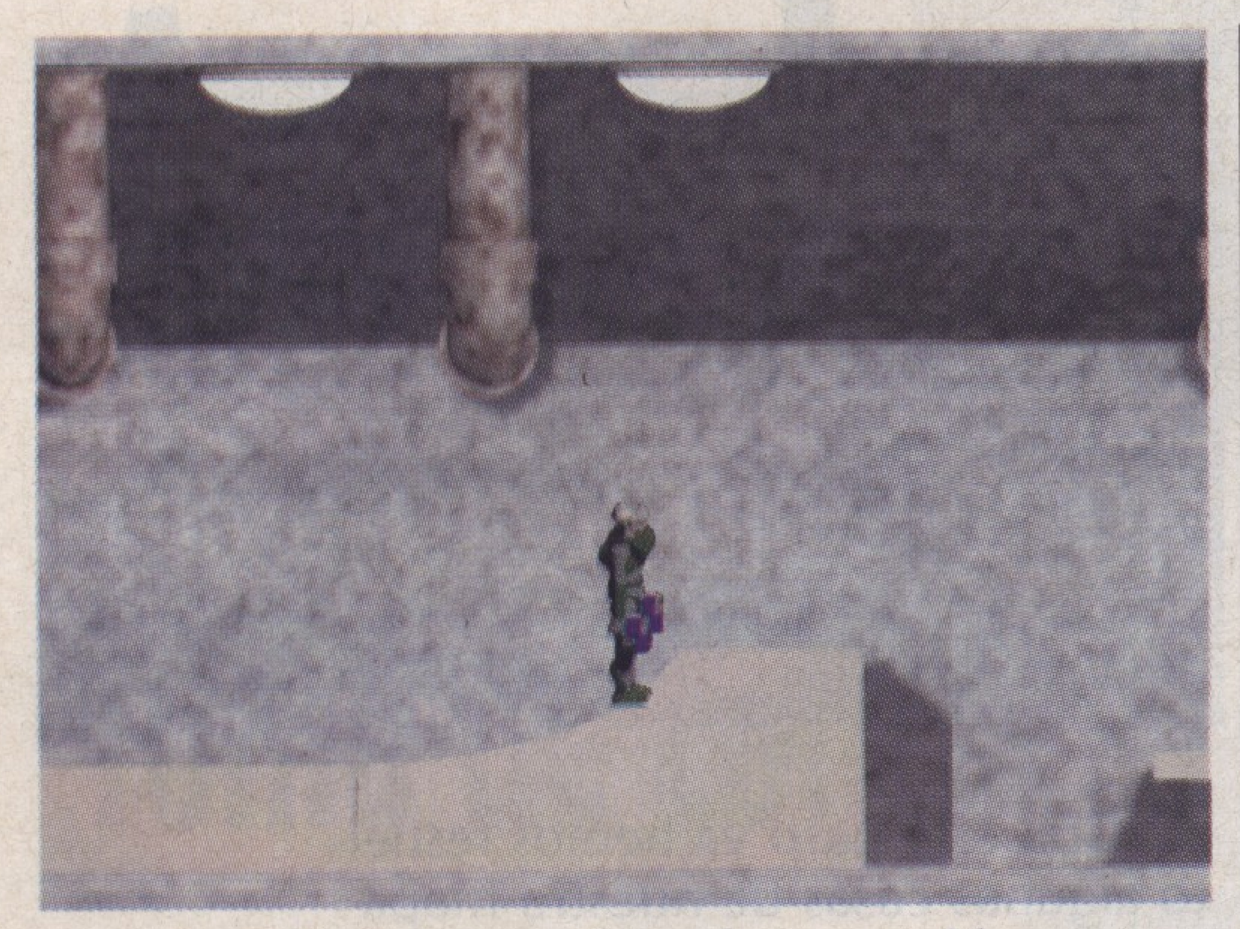
El método usado para el control del movimiento del personaje se basa en el efecto de gravedad

```
IF (KEY(_left));
   FOR(contador=0;contador<=
velocidad;contador++);
      IF(map_get_pixel(fichero,
grafico,(x-contador)/2,y/2==
color_pared)
         x-=contador-1;BREAK;
      END
   END
```

En esta primera parte comprobamos que no vamos a tocar ninguna pared con ese movimiento y, si así lo hiciera, avanzar todo lo que sea posible hasta ella. Nos queda todavía un paso más antes de dejar que el personaje avance sin más, remontar un objeto:

```
//si contador es mayor que
velocidad es que el bucle ha terminado sin encontrar ninguna pared que
se interponga al movimiento

IF(contador>velocidad)
   //iniciamos el bucle para comprobar pixel a pixel si encontramos
suelo
      FOR (contador=0;
contador<=limite;contador++);
```



Captura de nuestro ejemplo en acción.

```
        IF(map_get_pixel
(global_fpg,2,(x-velocidad)/2,
(y-contador)/2)
==color_suelo)
            //si lo encuentra pasa-
mos las coordenadas concretas y
paramos el bucle
               y-=contador;BREAK;
            END
         END
      x-=velocidad;
   END
```

Este pequeño fragmento calculará si hay posibilidad de remontar el terreno, no sin antes comprobar que no se haya chocado con una pared, comprobando simplemente que el contador anterior acabó sin ser interrumpido. Después, se comprueba que el sitio de destino donde vamos a ir tiene suelo en ese punto y en puntos verticales superiores al mismo, tantos como se le permitan con la variable límite, cortando el bucle si encuentra el suelo. Así controlaremos tanto si hay que remontar como si no, pues se empieza en la misma coordenada con la que se comenzó el bucle, haciendo un *break* en el mismo si lo encuentra.

Definimos una variable llamada dirección para los movimientos del personaje

Ya sólo nos queda controlar las animaciones del personaje para que, mientras se mueven sus coordenadas, presente los *frames* correspondientes según sea la dirección adecuada al caso.

```
      //ahora gestionamos los gráfi-
cos de las animaciones
    SWITCH(direccion)
      CASE 1: //izquierda
        graph++; flags=1;
        IF (graph>7) graph=1; END
      END
      CASE 2: //derecha
        //hacemos la animación de
giro
        FROM GRAPH=11 TO 8
STEP -1;
          FRAME;
        END
        //y ponemos el grafico inicial
espejado
        GRAPH=1; FLAGS=1;
      END
    END
```

Para los movimientos del personaje hemos definido una variable llamada dirección que guardará la última dirección, es decir, la del frame anterior. Esta variable es especialmente útil para el movimiento del personaje y nos va a permitir elegir la animación adecuada en cada momento.

Finalizando nuestro juego podemos incluirle la posibilidad de salto. Gracias a lo programado para el efecto de gravedad, sólo debemos ocuparnos de la primera parte del proceso, es decir, de ascender al personaje exactamente a la inversa que con la gravedad positiva; si antes era con aceleración constante, ahora con deceleración constante, siendo el impulso inicial el mayor, hasta que la fuerza de la gravedad termine venciéndola, la iguale a 0, e inicie su descenso hasta el suelo o plataforma más próxima.

En una parte inicial vamos a programar la lectura de la tecla *space*, que pondrá la gravedad en valores negativos, en este caso *–15*, sólo cuando estemos en el suelo.

```
  IF (KEY(_space))
    //si estamos en el suelo saltar
    IF (MAP_GET_PIXEL(fichero,
grafico,x/2,y/2==color_suelo)
      gravedad=-15;
    END
  END
```

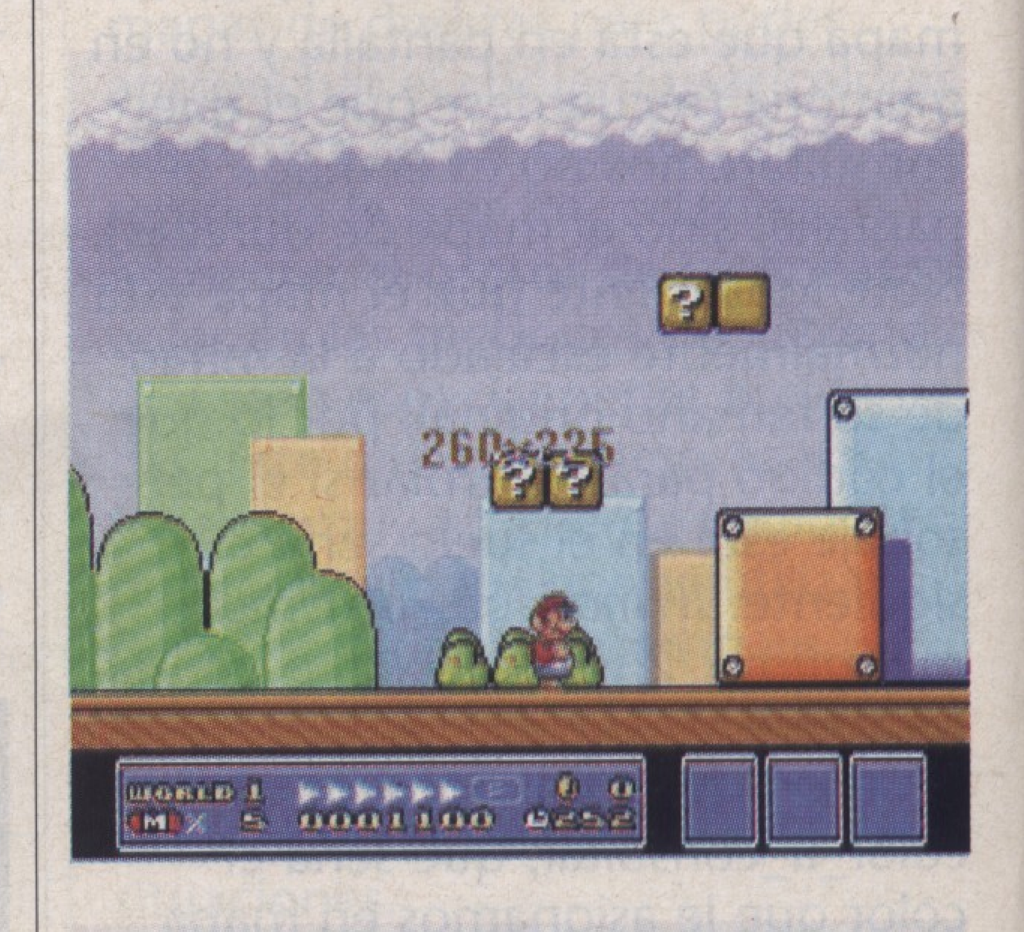
Además, una parte final que ascienda al personaje como la siguiente, en la que provocamos una deceleración, de la misma forma a como procedíamos con la gravedad, sumándole el valor de la gravedad (en este caso negativo), con lo que *y+=gravedad* sería por ejemplo *y+=-15*, no siendo válido en este caso *y-=gravedad* puesto que provocaría el efecto contrario, aunque también se podría hacer de la siguiente forma: *y-=abs(gravedad)*, con lo que obtendríamos el valor absoluto de la gravedad.

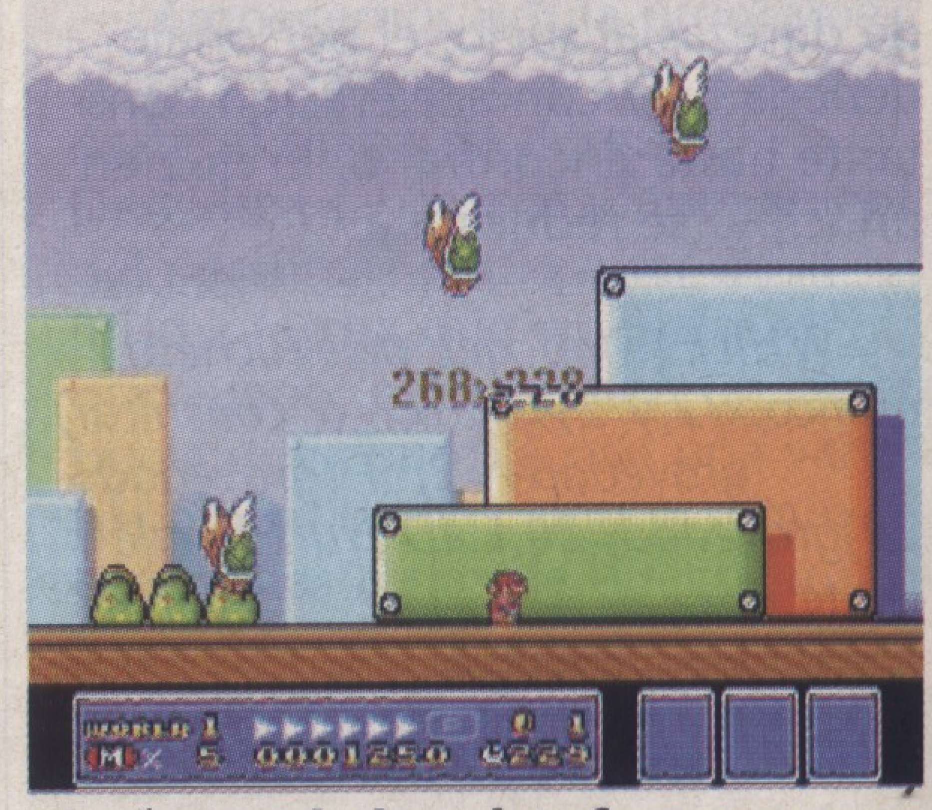
```
  IF (gravedad<0);
    //como el valor es negativo
debemos sumarle las coordenadas a
y, y no restársela pues provocaría el
efecto contrario
    y+=gravedad;
    gravedad+=2;
      IF (gravedad>0) gravedad=0;
END
  END
```

Los últimos retoques

Con este artículo deberíais ser capaces de programar correctamente un mapa de durezas, no obstante, os damos una serie de consejos finales para atar cabos.

En primer lugar, si miráis el código fuente que acompaña al ejemplo notaréis que la línea que tenemos para la detección del color en el mapa de durezas es sensiblemente diferente a la que hay en el artículo. Eso es debido a que los gráficos de la animación del personaje tienen su punto de control 0 en el centro (por defecto en todos los gráficos DIV). El punto de control 0 es el que DIV utiliza como referencia para las coordenadas del gráfico, de este modo, cuando nosotros le indicamos que queremos poner un gráfico en las coordenadas 100x100, en realidad en esas mismas coordenadas estará el centro de nuestro gráfico a menos que lo editemos. Esto hace que, en nuestro juego, cuando pasemos las coordenadas del personaje al *map_get_pixel* como *x,y,* en realidad estaremos enviando el centro del gráfico. Ante este problema tenemos varias soluciones: ponerle el punto de control 0 a todos y cada uno de los gráficos de las animaciones en los pies o tomar la distancia que existe entre el cen-





El género de las plataformas goza de una gran historia.

tro y los pies y sumárselas a las coordenadas en el momento de pasárselas a *map_get_pixel* de la siguiente forma:

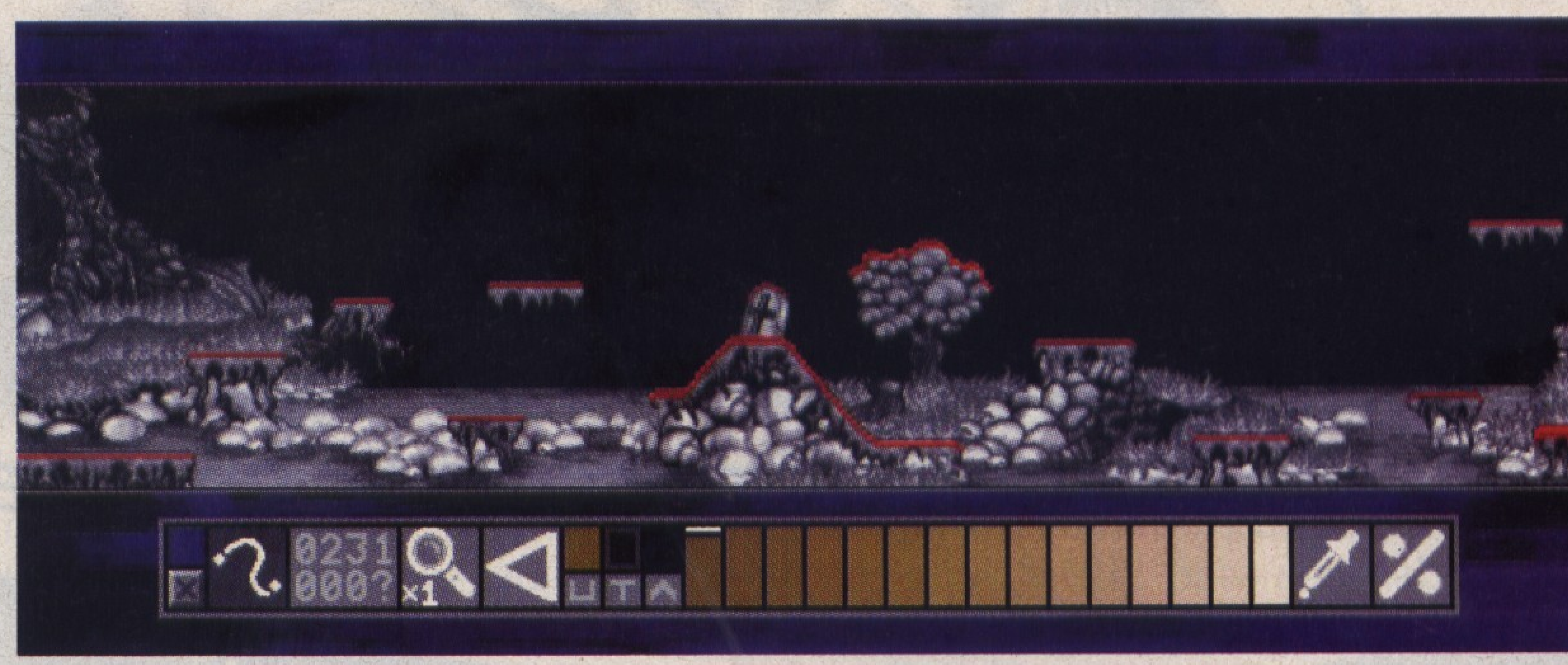
MAP_GET_PIXEL (fichero,grafico, x/2,(y+distancia_centro_pies)/2)

Con lo que conseguimos ahorrar tiempo al no tener que editar manualmente todos y cada uno de los puntos de control de los frames de cada animación de los personajes del juego.

También es posible que al pasar a escala de grises nuestro mapa del juego obtengamos un resultado totalmente ilegible en cuanto a color se refiere. Esto suele pasar cuando tenemos gráficos que usan una gama de colores muy concreta que nos ocupa la totalidad de la paleta. Para solventar este problema, al crear nuestra paleta para el juego la realizaremos en un programa de retoque fotográfico mediante un mapa que incluya el mapa de la fase, el personaje y un gradiente en escala de grises. Transformando el resultado a 256 colores obtendremos la gama de grises necesaria para hacer el mapa de durezas.

Como ya es habitual en el CD de la revista tenéis el código fuente del ejemplo comentado

Ya hemos comentado anteriormente que el proceso de detección de los mapas de durezas no es exacto. Debemos tener en cuenta que, al tener que dividir las coordenadas por la mitad, pueden darnos valores que, traducidos en el mapa de durezas, no correspondan exactamente con el que teníamos en el mapa real. Así, si en alguna parte del terreno del juego los personajes o enemigos caen sin remedio debido a esto, tendréis que modificar el mapa de durezas y aumentar el grosor de las líneas de color en aquellas partes que lo requieran.



Ejemplo de un mapa de durezas.

El ejemplo

En el CD que acompaña a la revista podéis encontrar el ejemplo de código fuente total de donde han salido los fragmentos que hemos comentado a lo largo de este artículo. Si os surgen dudas, mandadlas a la dirección de correo *migens@teleline.es* y se os responderán todas las consultas de forma individualizada y en el menor tiempo posible. También podéis mandar ejemplos y sugerencias sobre temas relacionados con esta sección.

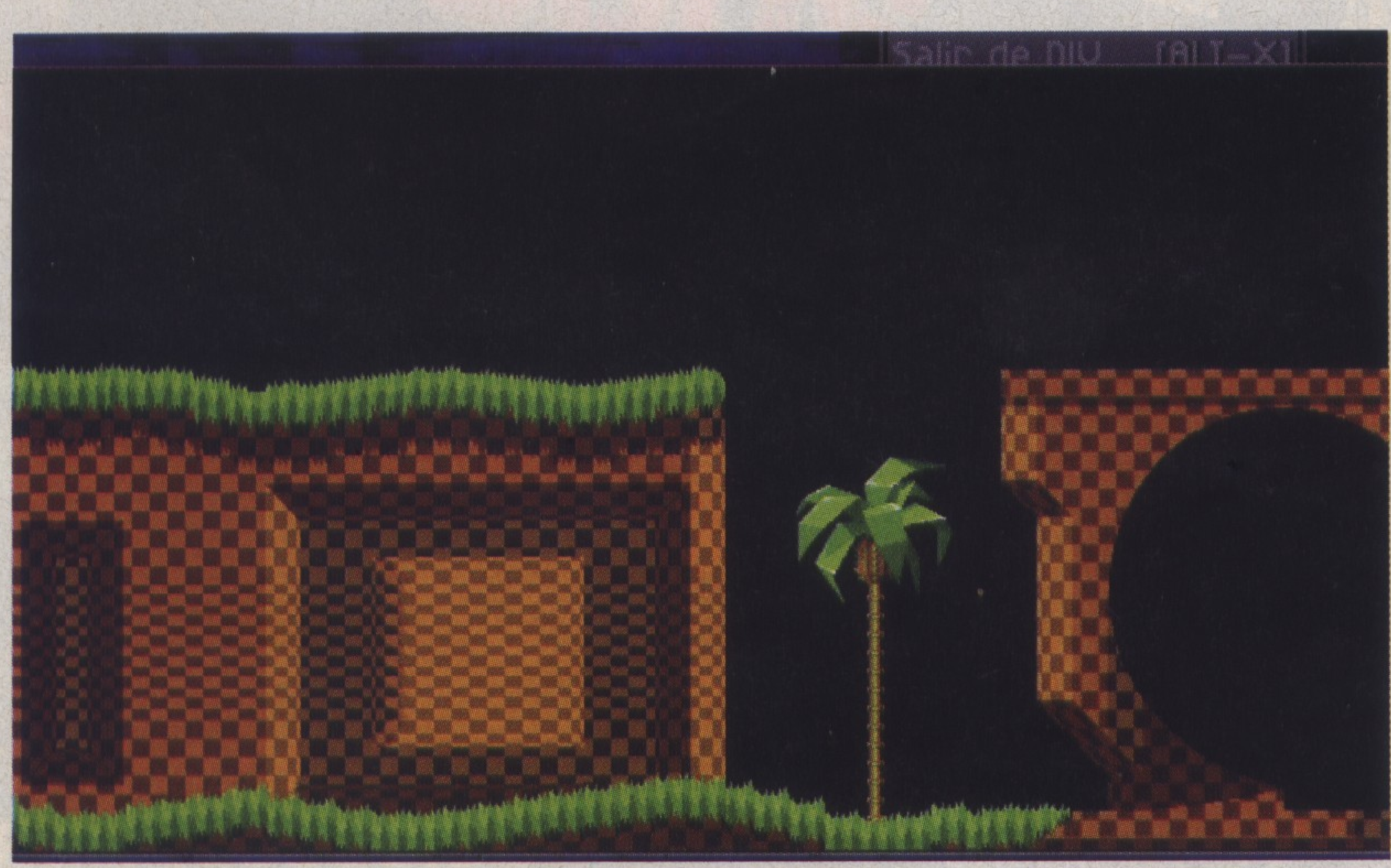
Para la ejecución del ejemplo debéis instalar los archivos en el mismo directorio y a ser posible en *c:\div2\divmania* o en su defecto modificad la ruta en los accesos a los archivos, en este caso en la carga de los dos fpg del juego.

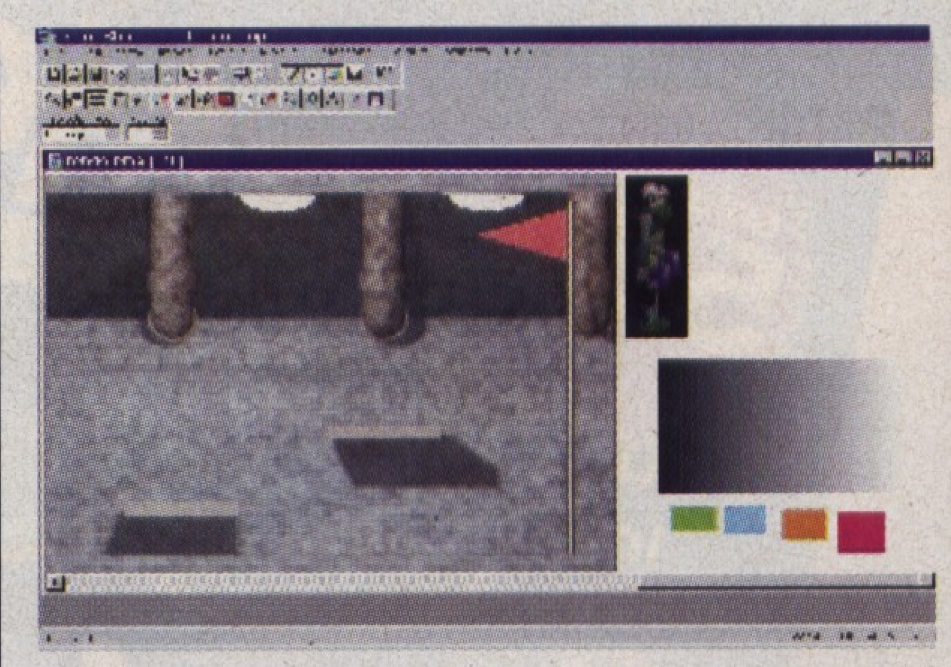
Agradecer, finalmente, la colaboración de J.M. Rosa Moreno por sus gráficos.

Dudas de los lectores

Respondemos ahora a una consulta de un lector que nos pregunta lo siguiente:

Al crear mis juegos suelo tener problemas con los colores y no sé cómo encajar en la misma paleta los colores necesarios para que mis gráficos no "muten", dándole un aspecto nada aceptable.





En busca de la paleta perfecta.

Respuesta

El problema de los colores, al igual que el de la gestión de memoria, requiere especial atención en DIV. El problema que nos planteas tiene varias soluciones. Cuando lo que realmente se desea es extraer la mejor paleta de entre todos los colores de los personajes que van a salir en la misma pantalla podemos usar programas especializados, como el Pallete Works, que ha sido comentado en ediciones anteriores de ***DIVmanía*** o, sencillamente, utilizar un programa de retoque fotográfico y realizar las siguientes acciones. Abrimos un mapa en 16 bits de color y vamos introduciendo todo los objetos, personajes y decorados del juego, exclusivamente aquellos que vayan a compartir paleta en un mismo momento puesto que los objetos de distintas fases pueden tener distintas paletas sin ningún problema. No importa que los gráficos se toquen o que estén montados sobre el decorado (en el caso de que no sean muchos objetos). Realizamos una conversión a 256 colores del resultado y lo abrimos con DIV aceptando la nueva paleta. Finalmente, cargamos los mismos objetos desde sus ficheros o *maps* correspondientes con un resultado visiblemente mejorado.

José Antonio Migens Gómez (Vital)
migens@teleline.es

Modos 7

Un paso más en el camino

Comenzaremos nuestro camino hacia las 3 dimensiones con los modos 7 o modos de plano abatido. Con estos modos podemos realizar nuestros primeros juegos en 3D sin mucho trabajo.

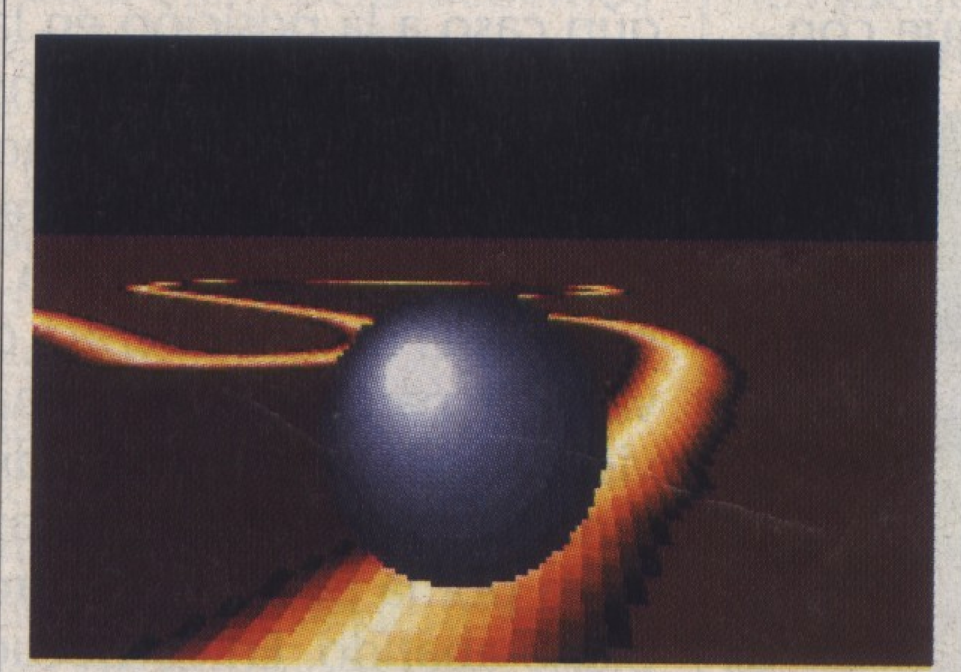
Lo primero que vamos a hacer es dejar muy claro qué son los modos 7. Se trata de una primera aproximación a los entornos completamente tridimensionales. Y decimos aproximación porque, aunque aparentemente creamos que tenemos ante nosotros un mundo tridimensional, existen aún muchas limitaciones, muchas de ellas remediadas en el modo 8 que veremos en un futuro. No obstante, nos servirá para familiarizarnos con el mundo de las tres dimensiones.

Una imagen en modo 7 consiste en un plano abatido haciendo las veces de suelo o techo de un escenario en tres dimensiones, donde los distintos objetos se sitúan en dicho plano como *sprites* o imágenes planas y no como figuras 3D auténticas. Dichas imágenes varían su tamaño para hacer el efecto de perspectiva. Un ejemplo de esta técnica lo podemos ver en juegos comerciales como el famoso Mario Kart o los que acompañan al CD de DIV: Speed for Dummies y Soccer.

Aunque, en apariencia, podamos pensar que tenemos un mundo en 3D, ésto es muy relativo

Podemos movernos a través de dicho plano abatido en cualquier dirección, pero todos los objetos deben estar sobre dicho plano. La única capacidad de desplazarnos en altura se produce en la cámara desde la que vemos la escena, que puede desplazarse arriba y abajo, así como cambiar su ángulo visión. Veamos una enumeración de todas las características principales que tiene este tipo de escenas:

- **Imagen de plano abatido:** es la imagen que se extenderá a lo largo de todo el plano abatido. Esta imagen deja de mostrarse a partir de la línea horizontal llamada horizonte.
- **Imagen de fondo:** es la imagen que se muestra más allá del plano abatido y por encima del horizonte si dicho plano abatido se muestra como un suelo o por debajo si se muestra como un techo.
- **Cámara:** es el punto desde el cual el jugador observa la escena. Puede estar asociada a un objeto en la propia escena de forma que percibamos constantemente al mismo desde su parte posterior. Variando la posición de la cámara es como podremos desplazarnos por la escena como si de un entorno tridimensional se tratase.
- **Altura:** es la distancia que separa el plano abatido (suelo o techo) de la posición de la cámara. Una altura positiva indica que el plano actúa como suelo y negativa como techo. Variando dicha altura podemos realizar efectos diversos como inmersiones en líquidos.
- **Distancia al proceso:** es la distancia a la que se sitúa la cámara por detrás del objeto al que sigue con la cámara.
- **Horizonte:** es la línea donde el plano abatido deja de extenderse al infinito. Se corresponde con el horizonte terrestre y, variando su altura, podemos crear el efecto de "mirar hacia arriba" y "mirar hacia abajo".
- **Focal de la cámara:** se corresponde con el ángulo que abarca el focal de la cámara. Si su valor es alto, los objetos se verán más cercanos.



Ventana de modo 7 con la cámara siguiendo a un proceso.



Ventana de modo 7 con la cámara en el proceso principal sin imagen asociada. Podemos volar sobre el escenario.

Veamos cómo podemos utilizar todas estas características en DIV.

Modos 7 en DIV

Los modos 7 se utilizan en DIV de una forma parecida a la de las ventanas de scroll. Es por ello que la estructura principal de este modo le puede ser muy familiar. Al igual que en éstos, los modos 7 se pueden representar en hasta 10 ventanas, numeradas de 0 a 9. Dichas ventanas ocuparán una región rectangular determinada de la pantalla que se definirá con el ya conocido método *define_region*. Para representar dicho modelo en DIV, disponemos de una estructura global llamada *m7*, que contiene todos los datos necesarios para modificar y caracterizar nuestra ventana de modo 7, así como una función *start_mode7()*, que nos permite establecer los valores iniciales de nuestra ventana.

Debemos tener en cuenta que las imágenes de fondo y de plano abatido deben encontrarse en el mismo fichero y, además, es recomendable que así sea con el resto de procesos que utilicemos en una misma ventana.

Para crear una ventana, en primer lugar debemos iniciarla. Para ello, debemos, como dijimos ante-

riormente, determinar la región de la pantalla donde aparecerá nuestra ventana de modo 7 mediante la función *define_region()*, aunque si deseamos que ocupe la pantalla entera, no es necesario. Este último caso se dará en la mayoría de las ocasiones, si bien en otros casos como los juegos de dos jugadores partidos (Speed for Dummies), convendrá definir regiones de otro tamaño.

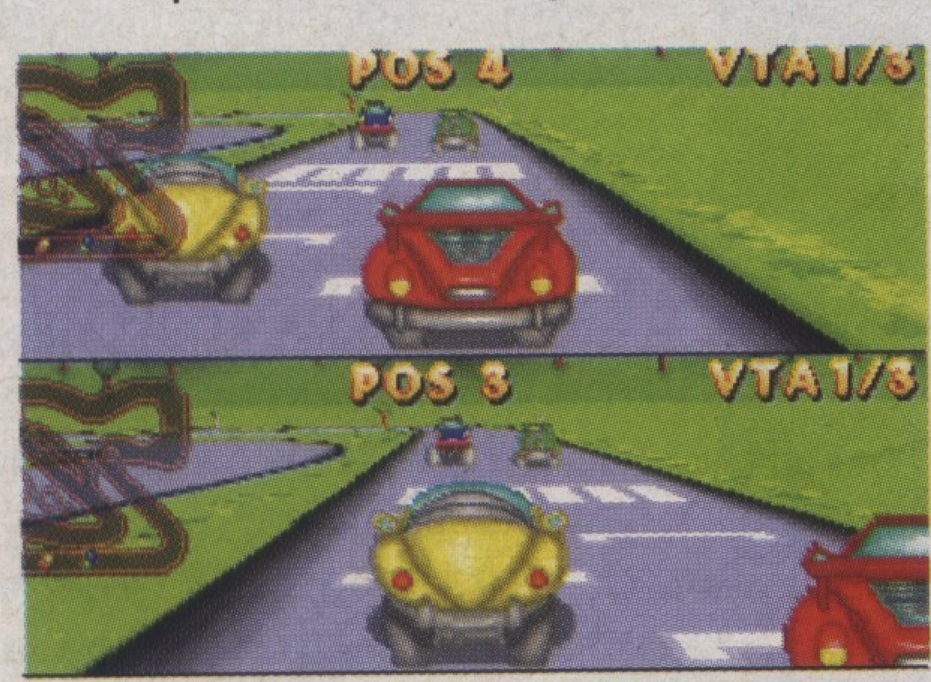
Definida o no la región, debemos cargar los correspondientes mapas que utilizaremos como plano abatido y de fondo (si existiere) mediante la función *load_fpg()*. Si hemos abierto alguna imagen con *load_map()*, se considerará como si se encontraran en el fichero 1. Ya estamos pues en disposición de iniciar la ventana de modo 7. La sintaxis que debemos seguir es la siguiente:

Start_mode7(<nº de ventana>, <fichero>,<imagen abatida>, <imagen de fondo>,<nº de región>, <altura del horizonte>);

Donde el número de ventana se referirá al índice de 0 a 9 que representará a nuestra ventana; fichero, el índice del fichero de imagen donde se encuentran los mapas abatido y de fondo; imagen abatida, el índice dentro del fichero de la imagen abatida; imagen de fondo, el índice dentro del fichero de la imagen de fondo; número de fondo es el valor devuelto por *define_region()* o 0 si deseamos que ocupe la pantalla completa; altura del horizonte es un número que nos indica dicha altura, que posteriormente analizaremos con detenimiento.

Debemos definir una cámara con la que mostrar la imagen de modo 7 para acabar de definir la ventana

Ya tenemos creada la ventana, si bien esto no es suficiente para mostrar ya las imágenes. Antes de nada, debemos terminar de definir los parámetros de nuestra ventana. Para ello se utiliza la estructura global *m7*, que es una tabla de 10 valores, cada uno representando al índice de la ventana de modo 7 correspondiente. Al igual que con



El juego Speed for Dummies en acción.

las ventanas de scroll, para referirnos a cualquiera de los parámetros de la ventana, debemos seguir la siguiente sintaxis:

M7[índice].parametro

Por cuestiones de simplicidad, si vamos a utilizar una sola ventana, podemos referirnos a *m7[0]* simplemente como *m7*.

Para terminar de definir la ventana, debemos definir una cámara con la que mostrar la imagen de modo 7. Para ello se utiliza la variable miembro de la estructura *m7* denominada *camera*, a la que debemos asignar el código identificador correspondiente al proceso que deseamos seguir. Si no queremos que siga a ningún objeto concreto, podemos asignarle en la secuencia principal del código la siguiente sentencia:

M7.camera = id;

que asigna el valor del identificador del proceso principal. Podemos crear una pequeña aplicación que nos muestre una ventana en modo 7. Aunque no podamos aún desplazarnos por ella, nos servirá para ilustrar todos estos conceptos descritos anteriormente:

```
PROGRAM ejemplo;

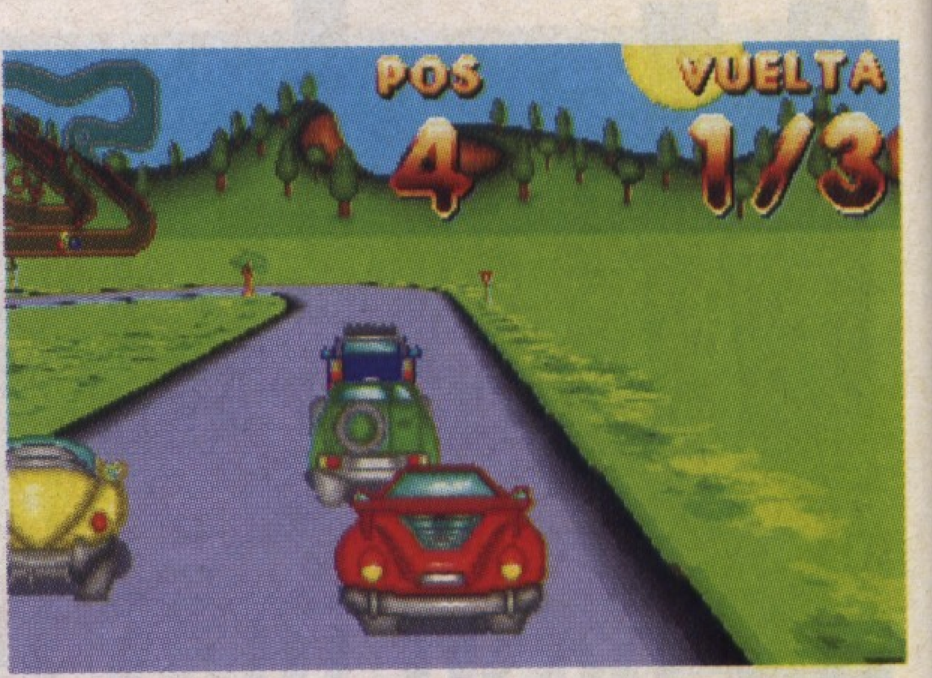
PRIVATE
fichero;

BEGIN
fichero = load_fpg("fichero.fpg");
start_mode7(0,fichero,1,0,0,64);

m7.camera = id;

LOOP
FRAME;
END
END
```

Para poder desplazarnos por el escenario, debemos tener en cuenta una cosa. Las coordenadas x e y de cualquier proceso que se muestre en una ventana de modo 7 están referidas al plano abatido, y en ningún caso a la posición en la pantalla. La variable z no tiene ningún significado especial a menos que dos imágenes se encuentren dentro de la ventana en la misma posición, caso en el que se utilizará dicha variable para saber qué imagen se visualiza. La única profundidad por tanto definida para este caso viene dada por la posición dentro del plano abatido. En consecuencia, podemos variar dichas variables x e y para movernos, sin embargo, es



De nuevo el juego Speed for Dummies, donde podemos jugar con dos jugadores creando dos ventanas de modo 7 donde se encuentran los mismos procesos.

más natural utilizar la variable *angle* para girarnos y *advance()* para movernos en la dirección en la que estamos mirando. Debemos recordar que si utilizamos estos parámetros propios de los procesos, también se encuentran en la secuencia principal de nuestro programa, ya que a todos los efectos se considera como otro proceso más.

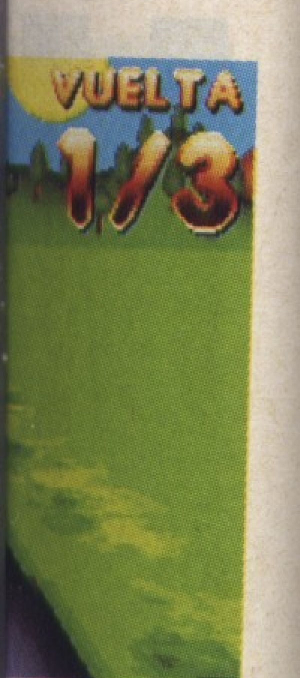
Vamos a utilizar los cursores para movernos por el escenario. Para ello debemos añadir las siguientes sentencias dentro del bucle de *FRAME:*

```
LOOP
  IF (key(_right)) angle-=
2000; END
  IF (key(_left)) angle+=
2000; END
  IF (key(_up)) advance(5); END
  IF (key(_down)) advance(-5);
END
    FRAME;
  END
END
```

Si ejecutamos ahora, esto se parece mucho más a un entorno tridimensional. Pero aún nos falta poder desplazarnos y mirar en torno al eje z o de altura. Para ello podemos utilizar los siguientes parámetros de la estructura *m7*:

– *Height* o altura de la cámara. Un valor positivo sitúa la cámara por encima del plano abatido y un valor negativo lo sitúa por debajo. Su valor por defecto es de 32, es decir, muestra el plano abatido como un suelo. Si lo variamos podemos hacer efectos de salto, agacharse, y un efecto que posteriormente analizaremos, que es el de la inmersión en líquidos.
– *Horizon* o altura del horizonte: nos indica la posición relativa del horizonte en la pantalla. El valor por defecto es el que le indiquemos en la llamada a la función *strat_mode7()*. Variando su valor podremos hacer el efecto de mirar arriba y abajo.

Al igual que en el caso anterior, podemos probar experimentalmente



qué ocurre cuando variamos dichos valores asignándoles ciertas teclas y aumentando o disminuyendo estos valores en consecuencia. Para ello basta añadir unas líneas al estilo de las añadidas para los cursores.

Ya nos falta únicamente analizar dos miembros más de la clase, que no dejan de ser importantes, que son:

- *Focus:* identifica al foco de la cámara. Su valor por defecto es de 256, y puede oscilar entre 0 y 512. Para valores altos, nos acercaremos más a los objetos del centro de la pantalla.
- *Color:* si no colocamos ningún mapa como imagen de fondo como en el caso del ejemplo (para realizar esto basta enviar un 0 como parámetro), DIV pinta dicha imagen con el color indicado por este parámetro, que por defecto es 0, es decir, el color transparente de la paleta.

Añadiendo procesos

Hasta ahora sólo nos hemos desplazado por un plano abatido sin ningún objeto. Al igual que ocurre con la estructura *m7*, esto se realiza de forma semejante a la que utilizamos para crear procesos en las ventanas de scroll. Para hacer que aparezca un proceso en este tipo de ventanas, debemos indicárselo en la variable *ctype* de la siguiente forma:

Ctype = c_m7;

Una vez realizado esto, debemos tener en cuenta que las variables *x* e *y* están ya referidas al plano abatido, y que la variable z pierde todo su sentido.

En el caso de crear múltiples ventanas de modo 7, si deseamos que un mismo proceso aparezca en varias ventanas a la vez (juegos multijugador, por ejemplo), debemos hacerlo con la siguiente sintaxis:

Cnumber = c_x + ... + c_z;

Donde *x* y *z* simbolizan el número de la ventana donde queremos que aparezca dicho proceso. Por ejemplo si queremos que aparezca en las ventanas 0, 1 y 3, debemos escribir *cnumber = c_0 + c_1 +c_3;*.

Como es evidente, al tratarse de un proceso tiene un gráfico asociado que se indica por la variable *graph*. Sin embargo, todo proceso tiene otra variable local llamada *xgraph* que sirve para asignar varias imágenes según el ángulo que forme respecto de la cámara. Para ello debemos seguir el siguiente proceso:



El juego Soccer que acompaña a DIV. Es una simulación sencilla del fútbol, si bien la principal limitación viene de la imposibilidad de desplazar el balón por encima del suelo.

Creamos una tabla global donde indicaremos en el primer elemento el número de imágenes que tendremos, y en el resto de elementos las imágenes que se mostrarán a partir de los 0 grados. Por ejemplo, podemos crear la siguiente tabla:

GLOBAL
Tabla[]=6,101,102,103,104, 105,106;

Asignamos la dirección de dicha tabla a la variable *xgraph*. Para ello, utilizaremos el operador OFFSET de la siguiente forma:

Xgraph = OFFSET tabla;

Ya podemos, por tanto, crear varios procesos en nuestra ventana de modo 7 y admirar el resultado. Es conveniente trastear y experimentar por vuestra cuenta para poder comprender a fondo el significado de cada una de las variables de esta estructura.

Destruyendo los modos 7

Es una pena, pero todo se acaba alguna vez. Quizá, en algún momento, queramos hacer desaparecer las ventanas de modo 7, como ,por ejemplo, cuando dejamos el juego para volver al menú. Para realizar esto disponemos de la función *stop_mode7()*, que recibe como parámetro el número de la ventana de modo 7 que deseamos destruir.

Hasta aquí bien, pero debemos tener en cuenta que todos los procesos que se hayan declarado como de tipo *c_m7* y que no se estén visualizando en ninguna otra ventana de modo 7 se destruirán durante este proceso.

Además de esto, debemos tener en cuenta que al igual que las ventanas de scroll, cualquier ventana de modo 7 se destruirá si cambiamos de modo de pantalla con la función *set_mode()* y, por tanto, también se destruirán todos los procesos asociados a toda ventana de este tipo.

Creando efectos sencillos

Este tipo de juegos tiene un gran número de posibilidades, vamos a ver un ejemplo en el que ilustraremos dos cosas importantes: en primer lugar, cómo asociar un objeto a una cámara, y en segundo lugar, cómo hacer un efecto de inmersión en agua variando la altura de la cámara.

Este efecto consiste en considerar el suelo como si de un líquido se tratara para poder sumergirnos en él, si simplemente deslizáramos la altura de la cámara hasta considerar el suelo como techo, veríamos de una forma antinatural la inmersión y no nos sentiríamos como si estuviéramos bajo el agua. Es por ello que en cada *frame* analizamos la posición de la cámara para ver si estamos por debajo del horizonte o por encima. Si es negativo, se realiza un fundido rápido de colores donde el azul predomina sobre el resto. Para ello utilizaremos la función *fade*, en la que conservaremos todos los tonos al 100%, excepto el azul, que colocaremos al 150%. Cuando volvamos a la superficie, recuperaremos el tono normal de la imagen colocando de nuevo todos los tonos al 100%. De esta forma, podemos aplicar cualquier tipo de filtro de luminosidad, saturación o color a cualquier tipo de imagen.

Si no colocamos ningún mapa como imagen de fondo, DIV pinta éste con el valor por defecto, es decir 0

Conclusión

Estos son todos los conocimientos que debéis tener para realizar cualquier juego en modo 7. Cualquier tipo de efecto, como utilizar suelo y techo a la vez, pasa por técnicas combinadas (en este caso se deben utilizar dos ventanas de modo 7 al mismo tiempo). Dejamos eso a la creatividad de cada uno. Para ver ejemplos prácticos de este modo os remitimos a los juegos de ejemplo Speed for Dummies y Soccer, aunque será más sencillo comenzar leyendo el programa de tutorial número 7, donde se ilustra perfectamente la rotación de varias imágenes asociadas a un mismo proceso. Para cualquier pregunta o sugerencia al respecto, podéis dirigiros a la siguiente dirección de e-mail: *trinidad@arrakis.es*.

Pablo Trinidad

Creando un generador de código (I)

Seguimos aprendiendo

Actualmente nos vemos rodeados por aplicaciones que nos permiten realizar programas casi sin teclear ninguna línea de código. A estas aplicaciones de diseño rápido o RAD nos referiremos en este artículo, donde intentaremos emularlas para crear nuestros juegos de DIV con un entorno completamente visual.

Continuando con la creación de aplicaciones externas para DIV, vamos a crear un pequeño generador de código en el que podremos construir el esqueleto de un juego cualquiera introduciendo tan sólo algunos parámetros de forma sencilla. Un ejemplo de este tipo de aplicaciones puede ser DIV Generator o Creplat (aunque este último sea de propósito más específico al ser un generador de juegos de plataformas).

Vamos a analizar, paso a paso, la creación de este pequeño programa y para ello utilizaremos Visual C++ 6, aunque bien podría haberse escogido cualquier otro compilador. Hemos escogido este entorno porque, en primer lugar, está basado en el estándar C/C++, que es el lenguaje que utilizamos en el desarrollo de dlls y, por tanto, nos resulta más familiar, y, además, utilizaremos pocas capacidades específicas de este compilador. Tan sólo utilizaremos el editor de recursos, que viene a ser semejante a otros como el Borland C++.

Pasemos pues a la creación de nuestro generador de código.

Figura 1. El AppWizard en acción. Esta aplicación nos permite generar código de forma rápida, al igual que nuestro programa.

Eligiendo las capacidades básicas

En primer lugar, debemos escoger cuáles van a ser las capacidades que vamos a introducir en nuestro programa. Para comenzar, escogeremos tan sólo algunas capacidades básicas, que fácilmente podremos ir ampliando a nuestro antojo.

En nuestro caso tomaremos estos 3 apartados:

- **Datos personales:** nos servirá para autentificar nuestro programa introduciendo nuestros datos personales en la cabecera como comentario. Serán el nombre del autor, el nombre del programa (que utilizaremos también como nombre para la cabecera *PROGRAM)* y el año de producción.
- **Gráficos:** en este apartado especificaremos el modo de pantalla que utilizaremos en nuestro juego, así como el fichero de imágenes principal (índice 0) y el índice dentro del fichero que contendrá el fondo de pantalla que cargaremos al inicio del programa.
- **Ratón:** aunque no es un apartado esencial en cualquier programa, nos servirá para ejemplificar la creación de un proceso. Si escogemos la opción de que aparezca, deberemos señalar el índice que ocupa el gráfico del ratón dentro del fichero principal con índice 0.

Una vez realizado este pequeño esquema, podemos pasar a la creación en programa en sí.

Generando el esqueleto de nuestro programa

Visual C++ es un entorno que nos permite generar de forma rápida y cómoda el esqueleto de casi cualquier tipo de aplicación. Para ello utilizamos lo que se denominan *wizards*, que no es otra cosa sino un generador de código al estilo que vamos a desarrollar nosotros mismos. Es por eso que nos puede servir para hacernos una idea del objetivo que perseguimos con este programa, así como la estructura de éstos.

En primer lugar, y una vez abierto el compilador, debemos crear un nuevo proyecto. Para ello debemos elegir la opción New del menú *Archivo*. Nos aparecerá una imagen parecida a la de la Figura 1. Estamos ya en el *wizard* para aplicaciones. Como vemos hay muchos tipos de aplicaciones para generar. Nosotros optaremos por *MFC AppWizard (exe)*, es decir, crearemos un ejecutable para Windows basado en MFC, que no

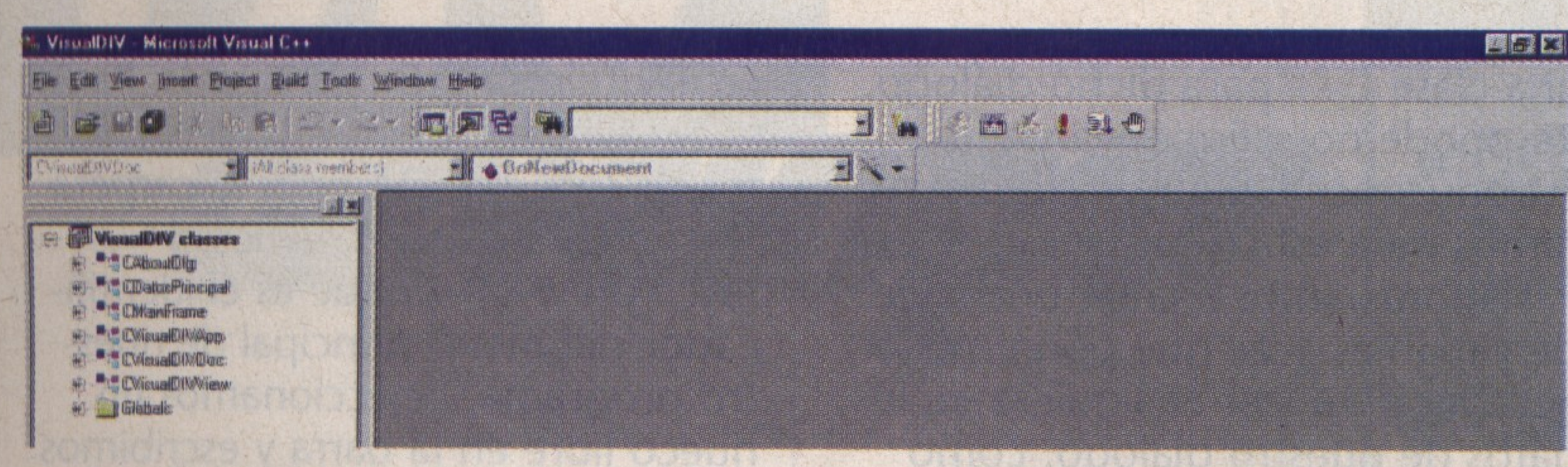


Figura 2. El entorno de Visual C++ 6 una vez generado el proyecto. A la izquierda se encuentra la ventana de proyecto donde podemos acceder al código y a los recursos.

es más que una encapsulación en clases de la API de Windows. Una vez realizado, debemos introducir el nombre de nuestro proyecto, así como el directorio donde lo guardaremos. Hemos llamado al proyecto VisualDIV, si bien podemos llamarlo de cualquier otra forma. Estamos ya dispuestos para elegir las opciones de nuestro proyecto. Nos desplazaremos por varias pantallas donde debemos elegir las siguientes opciones:

- **Paso 1:** debemos cambiar a Single Document (SDI), ya que no abriremos múltiples ventanas al mismo tiempo. No es estrictamente necesario, si bien simplifica en gran medida el código.
- **Paso 2:** sin cambios.
- **Paso 3:** desactivamos los controles ActiveX.
- **Paso 4:** eliminamos la opción *printing and print preview*, que nos introduce las opciones de imprimir y vista previa de los documentos. No lo necesitaremos. Además, debemos colocar a 0 el número de archivos recientes que deben aparecer en el menú *Archivo*.
- **Paso 5:** En cuanto al uso de las librerías MFC podemos elegir cualquiera de las dos opciones, si bien debemos tener en cuenta que si elegimos DLL compartida, debemos distribuir junto con el ejecutable la correspondiente librería dll de las MFC.
- **Paso 6:** Aquí ya tenemos creado el proyecto. Si queremos cambiar el nombre a alguna clase éste es el momento, aunque no suele ser demasiado conveniente.
  Podemos pulsar el botón de finalizado para crear definitivamente el código del esqueleto de nuestro programa.

Si hemos seguido bien todos los pasos tendremos en pantalla una ventana como la de la Figura 2. En la ventana de la izquierda, llamada *workspace*, disponemos de toda la información acerca del proyecto: clases, ficheros y recursos. Estos últimos comprenden todos los datos acerca de menús, diálogos y mapas entre otros. Antes de comenzar con el diseño de las distintas ventanas donde aparecerán los datos a rellenar, vamos a ejecutar lo que tenemos de programa para hacernos una idea del funcionamiento base. Para ello podemos hacerlo de dos formas:

- **Modo ejecución:** se elige pulsando el símbolo de exclamación en la barra de botones, pulsando *F5* o eligiendo la opción *execute* del menú *Build*. Simplemente genera y ejecuta nuestro programa al modo que lo haríamos desde la línea de comandos.
- **Modo depuración:** en este modo, podemos ejecutar paso a paso nuestro programa para detectar posibles fallos (utilizando para ello *F10* y *F11*) o colocar lo que se denominan *breakpoints* (pulsa *F9*) o puntos de ruptura donde el proceso de ejecución se debe parar para pasar a ejecución paso a paso. Para activar este modo debemos pulsar F5 o elegir en el menú *Build* la opción *Debug – Go*.

Ahora podemos ver el funcionamiento básico de nuestro programa. No es nada más que una ventana que no hace nada aparte de mostrar un diálogo de *Acerca de* e información en la barra de tareas.

Pasemos una vez conocido cómo ejecutar nuestro programa a la generación de las diferentes ventanas.

Creando la ventana de características básicas

En nuestro programa y por cuestiones de simplificación, vamos a crear un diálogo donde colocaremos las opciones discutidas al comienzo, y para cargarlo crearemos una opción de menú que se llame *Datos básicos*.

Para crear una nueva ventana de diálogo, debemos, en primer lugar, acceder al archivo de recursos. Para ello, utilizaremos la ventana de *Workspace* y accederemos a la pestaña de recursos. En esta pestaña debemos escoger el apartado *dialog*, y pulsando con el botón derecho del ratón insertaremos un nuevo diálogo. Veremos como nos aparecerá una nueva opción llamada *IDD_DIALOG1*. Debemos pinchar sobre ella y se nos abrirá la imagen con el diálogo. Es un diálogo con dos botones de aceptar y cancelar. Disponemos de una lista de controles que podemos insertar a la derecha (si no la vemos, basta con pulsar el botón derecho del ratón y elegir la opción *Controls*). Para hacernos una idea de cómo vamos a diseñar nuestro diálogo, podemos ver la Figura 3. Esta es una posible estructura, si bien puedes adoptar la que más te guste.

El proceso de creación de los distintos controles (botones, cajas de texto, paneles, etc.) es el siguiente:

Seleccionamos en la barra de controles el control apropiado, y pulsando sobre el diálogo lo colocamos a nuestro gusto, tanto en tamaño como en posición.

Pulsando sobre el botón derecho sobre cualquier control aparecerá la opción de propiedades, donde introduciremos las características de cada uno de los controles. Para que permanezca de forma permanente en la pantalla podemos pulsar el botón de la esquina superior izquierda.

Seleccionamos un nombre descriptivo para el recurso (nombre en mayúsculas), así como el título adecuado que aparecerá en la pantalla.

Visual C++ es un entorno que nos permite generar el esqueleto de una aplicación

Seleccionamos las propiedades de cada uno de los controles.

Seguiremos este proceso para la creación de los siguientes controles:

- Un panel de datos personales, donde aparecerán los siguientes controles:
  - Dos cajas de edición donde introduciremos los datos de nombre del autor y nombre del programa.
  - Dos controles *Static* de texto estático que contendrán el texto correspondiente a cada una de las cajas de texto anteriores, para indicar a qué se refiere cada una.
- Otro panel de gráficos, donde aparecerán los siguientes controles:
  - Una caja combinada donde introduciremos los datos de cada uno de los modos de vídeo que posee DIV. Para introducirlos, debemos escribir cada uno de los elementos que deseemos colocar en la caja en la pestaña de datos de la ventana de propiedades, y para cada nueva línea debemos pulsar *Ctrl+Enter.*
  - Una caja de edición donde introduciremos el número del

Figura 3. Esta es la estructura que va a seguir nuestro diálogo.

índice de la imagen de fondo a cargar. Para indicar que sólo se acepten números como caracteres, debemos especificar la propiedad *Number* en el menú de propiedades.

- Una caja de edición donde aparecerá el nombre del fichero de imágenes. Podremos abrir una ventana de abrir archivo para elegir el nombre del fichero. Para ello, crearemos un botón con el título de cargar.
- Los textos estáticos correspondientes a cada apartado para que el usuario pueda distinguirlos correctamente.

Los diálogos modales son mucho más sencillos de crear

- Un último panel para el ratón donde aparecerán los siguientes controles:
  - Una caja de verificación o *checkbox* para indicar que queremos utilizar esta característica.
  - Una caja de edición donde, al igual que con la imagen de fondo, introduciremos el número de la imagen que queremos cargar como cursor.
  - Los textos estáticos correspondientes.

Dando forma a nuestro diálogo como código

Ya tenemos diseñada nuestra ventana de diálogo. Ahora tan sólo nos queda asignarle una clase de C++ para poder utilizar todas sus propiedades. Para ello utilizaremos lo que se denomina *ClassWizard*, un asistente muy cómodo donde podremos asignarle a cada diálogo una clase y en ella definir los datos que introduzcamos en los distintos controles, como si de variables se trataran. Para arrancarla, basta con pulsar dos veces en la caja de diálogo o en el botón correspondiente de *ClassWizard* (indicado por una varita mágica), y nos preguntará inmediatamente si queremos crear una clase C++ para dicho diálogo. Respondemos afirmativamente e introducimos el nombre que queramos para esta nueva clase.

Ahora vamos a crear una serie de variables a las que podremos acceder para leer cualquiera de los datos de nuestro diálogo, como puede ser el nombre del autor, etc. Para ello elegiremos la pestaña correspondiente a las variables, llamada *member variables*. Nos aparecerá una lista de los identificadores de cada control. Haciendo doble clic sobre ellos se nos abrirá una ventana, donde podremos elegir el nombre de la variable que le hará referencia, así como el tipo de dato asociado. Podemos ver en la Tabla 1 los nombres de las variables asociadas a cada identificador, así como su tipo de dato.

Una vez realizado esto, estamos en disposición de utilizar dichas variables. Pero debemos tener en cuenta una cosa: el valor en cada instante de los contenidos de las variables no se refleja a menos que se lo indiquemos específicamente. Es decir, si introducimos en la caja del nombre del autor nuestro nombre, no aparecerá reflejado en la variable si no se lo indicamos implícitamente, y si variamos internamente el valor del nombre del autor, tampoco aparecerá en pantalla. Por ello, existe la función *UpdateData(BOOL tipo)*. Esta función se encarga de realizar la actualización de los valores de las variables. Si le pasamos como parámetro *true*, se cargarán los datos del diálogo en las variables, y si el valor es *false*, se grabarán los datos de las variables en los controles. Este proceso es muy importante, ya que es fundamental para poder trabajar con cualquier dato del diálogo.

Dando funcionalidad a nuestro diálogo

Una vez creado nuestro diálogo, ¿cómo podemos mostrarlo en pantalla asociado a una acción? Es muy sencillo. En primer lugar, vamos a crear una nueva entrada en el menú de nuestro programa cuyo título será *Datos básicos*. Para ello, utilizaremos de nuevo el editor de recursos. Abrimos la carpeta de menús y seleccionamos *IDR_MAINFRAME*, que es el identificador del menú principal de nuestro programa. Seleccionamos un hueco libre en la barra y escribimos en el menú de propiedades relacionado con dicho hueco, el título de nuestro menú. Desactivamos la opción de *Popup*, ya que no vamos a incluir ningún submenú dentro de esta opción. A continuación, le asociamos un identificador que, en nuestro caso, será *ID_MENUDATOSBASICOS*, aunque bien puede ser otro cualquiera. Ya tenemos una nueva opción en el menú.

Ahora, queremos que cada vez que se pulse dicha opción, se ejecute el diálogo. Para ello, debemos crear un evento asociado a dicha selección. Utilizaremos pues de nuevo *ClassWizard*. Buscamos en la pestaña del mapa de mensajes el identificador de nuestra opción de menú, y pulsamos en la ventana derecha donde pone *COMMAND*. De esta forma, crearemos una función *OnMenuDatosBasicos* donde implementaremos la funcionalidad de dicha opción. En nuestro caso, tan sólo queremos abrir la ventana de diálogo.

Existen dos formas de mostrar diálogos: los diálogos modales, que detienen la ejecución de la ventana que los llama hasta que se cierra dicho diálogo, y los no modales, que se abren como si de una aplicación independiente se tratara y, por tanto, puede visualizarse sin interrumpir la ejecución del programa.

Nosotros utilizaremos los diálogos modales, por ser mucho más sencillos de crear, como veremos a continuación. Para ello tan sólo necesitamos crear una instancia de la clase a que representa dicho diálogo, que en nuestro caso hemos llamado *CDatosPrincipal*. Una vez hecho esto, tan sólo debemos llamar al método de la clase *DoModal()* para que se muestre dicho diálogo. Por tanto, el código de dicha función será:

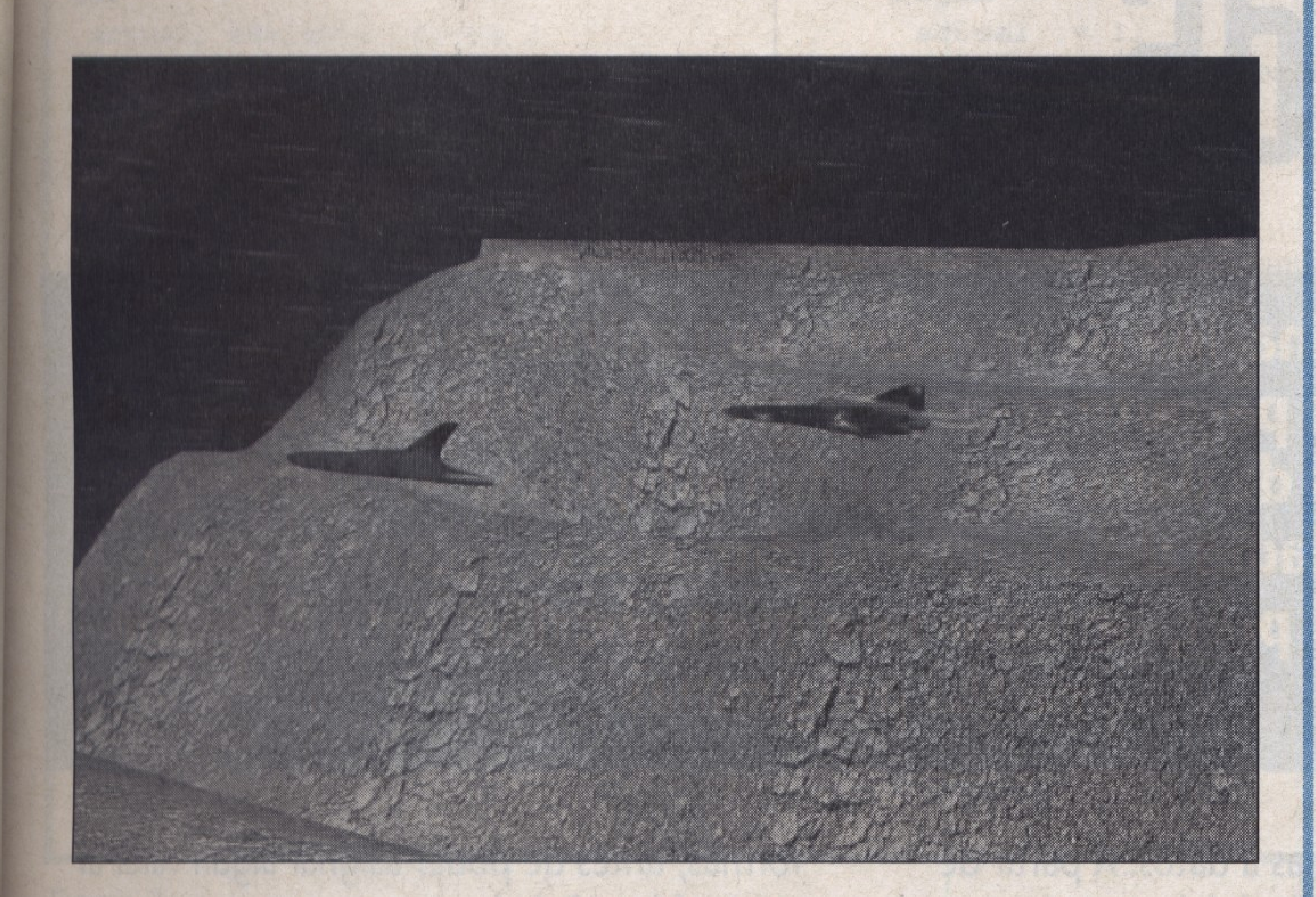
Tabla 1. Lista de los identificadores con sus tipos de dato y nombre correspondientes

Identificador	Tipo de dato	Nombre de la variable
IDC_CHECKRATON	BOOL	m_Raton
IDC_COMBOMODOSGRAFICOS	CString	m_ModoGrafico
IDC_EDITAUTOR	CString	m_Autor
IDC_EDITFONDO	Int	m_IDFondo
IDC_EDITFPG	CString	m_FPG
IDC_EDITPROGRAMA	CString	m_NombrePrograma
IDC_EDITRATON	Int	m_IDRaton

DIV developer

NÚMERO 8

Sumario

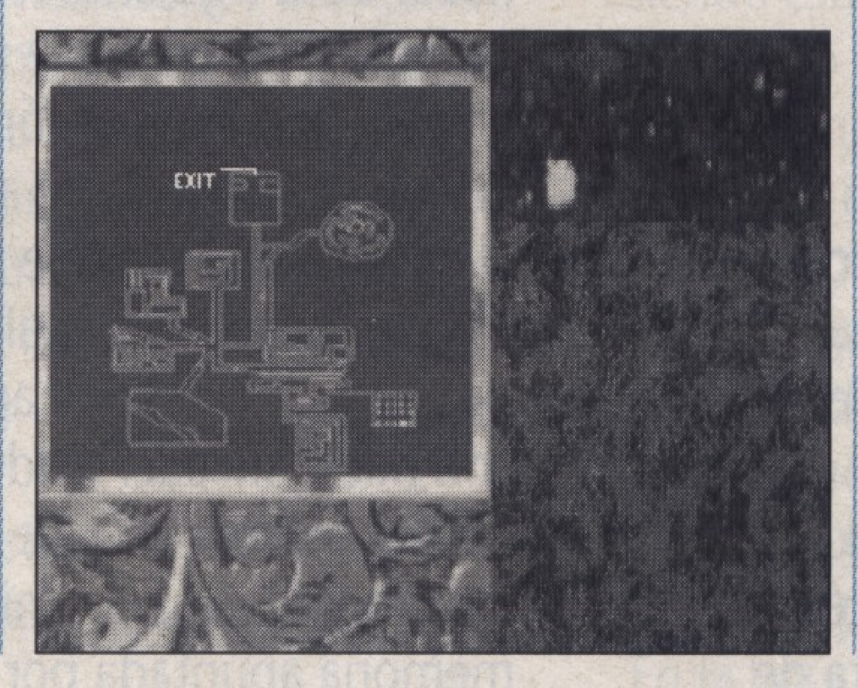


Curso de programación y concurso

Continuamos con nuestros cursos de programación habituales siguiendo los capítulos ya iniciados. Como siempre, esperamos que os sean de utilidad.
En cuanto a nuestro concurso, por supuesto sigue en pie con los mismos premios. Ya sabéis: 25.000 pesetas para el ganador y 20.000 para los otros dos juegos elegidos. Esperamos con impaciencia recibir vuestros juegos.
Aquí tenéis los ganadores de este número. Esperamos que os gusten. En este suplemento encontraréis un breve comentario del juego y un extracto de su código fuente que, no os olvidéis podéis encontrar entero en el CD-Rom si abrís los ficheros pertinentes. Seguro que la visión y el estudio de estos códigos os ayudará a mejorar algunos aspectos de vuestros propios trabajos de programación.

Vuestras sugerencias

Queremos haceros saber que todas las ideas, críticas o sugerencias para mejorar esta revista serán bienvenidas por parte de esta redacción. Si tenéis inquietud por algún tema que no tocamos demasiado, si consideráis que alguna sección de esta revista debe desaparecer o debe ser cambiada por otra, incluso si os animáis a proponernos artículos realizados por vosotros/as, que creéis pueden ser aprovechados por otros usuarios/as, no dudéis en poneros en contacto con nosotros. Todas las sugerencias serán atendidas y tenidas en cuenta. Os necesitamos para mejorar.
Para mandarnos los juegos o cualquier sugerencia sobre la publicación, podéis hacerlo por e-mail (gover@prensatecnica.com), correo normal o incluso entregarlos a mano en la dirección de la redacción:
Divmanía
C/ Alfonso Gómez, 42
Nave 1-1-2
28037, Madrid, España





Para los juegos del concurso

No os olvidéis meter vuestros datos completos cuando mandéis juegos al concurso. Lo decimos porque hemos recibido algunos en los que falta la ciudad, el teléfono o directamente la dirección entera. Para hacer más cómoda la tarea, en el CD que acompaña la revista hay una ficha que podéis incluir como fichero de vuestro juego, así no habrá problemas. Esta ficha se incluirá en la revista cuando hagamos otro especial de juegos DIV y será obligatoria para participar en el concurso.
Es vital que introduzcáis un fichero de texto que contenga la presentación de la historia, vuestras fuentes de inspiración, las características del juego, su instalación y, sobre todo, el código fuente.

Ficha Juegos DIV

Nombre del juego:
Autor (es):
Dirección:
Teléfono: E-mail:
Hablanos sobre tu juego (no seas conciso, puedes extenderte lo que quieras):
Descripción del juego:
¿Cuál es el objetivo del juego? ¿De dónde surgió la idea?
¿Qué es lo más destacado del juego según tu opinión?
¿Qué es lo menos? ¿Cuál es el objetivo del juego?
¿Qué problemas has encontrado en su desarrollo?
¿Quieres añadir algo más?
(Para contestar a estas preguntas usad la ficha que encontraréis en el Cd-Rom)

Destacamos

En nuestro CD incluimos los juegos que han resultado ganadores en este número y, cómo no, sus respectivos códigos para que veáis todo el proceso de creación. Además, unas cuantas demos y programas que nos han enviado los lectores. Si tenéis algo que pueda resultar de utilidad, enviárnoslo y lo meteremos en el próximo CD.

Punteros y listas

La combinación de punteros con estructuras de datos estáticas permite crear una gran variedad de nuevas estructuras de datos que pueden adaptarse, en tamaño, a las necesidades del programa durante su ejecución y que permiten al programador una gran libertad para el diseño de sus programas.

Todas las estructuras de datos que se han descrito en anteriores artículos tienen una característica común: quedan perfectamente definidas en tiempo de compilación, tanto en el tipo de elementos que pueden contener como en el número de éstos. Esta característica queda reflejada en la definición de estructuras de datos de tipo array. Los elementos de este tipo pueden ser tan complejos como se desee, de igual forma ocurre con el número de elementos del tipo base que pueden albergar (teniendo en cuenta las limitaciones de memoria), pero, una vez declarado, no se puede modificar ni su estructura ni su número de elementos.

Aunque es conveniente que el formato del tipo base de las estructuras sea estático, pues éste limita el tipo de elementos que albergará, no lo es tanto que esté determinado el número de elementos, ya que existen situaciones en las que o bien todos los elementos no se utilizan (toda la memoria reservada por el compilador no se utilizará con el consiguiente derroche de memoria), o bien no se ha reservado suficiente memoria, perdiendo el programa la generalidad que debería tener.

PUNTEROS

Para resolver el problema de la limitación de elementos que puede tener una estructura de datos, el lenguaje debería disponer de una serie de procedimientos predefinidos capaces de solicitar una cantidad de memoria adecuada para albergar a los nuevos elementos que se deseen insertar en la estructura de datos, e inversamente, otros que permitan liberar la memoria de aquellos elementos que se deseen eliminar. En esta nueva situación el acceso a la totalidad de los elementos ya no se puede realizar a través de variables definidas en el programa para cada uno de ellos, pues no se conoce, a priori, cuántos elementos formarán parte de la estructura, así, el lenguaje deberá permitir diferenciar entre datos y referencias a datos.

Cuando se utiliza una variable se está accediendo realmente al contenido de la posición de memoria asociada a dicha variable, así, la diferencia entre variable y contenido de ésta no existe, por eso no es posible distinguir entre datos y referencias a éstos.

Los lenguajes de alto nivel disponen de punteros que permiten realizar la distinción entre datos y referencias a datos. A partir de este punto, y hasta el final del presente artículo, se describirán los punteros utilizando el lenguaje Turbo Pascal, aunque la descripción no perderá generalidad respecto de otros lenguajes salvo en la sintaxis.

Un puntero no es más que una variable que contiene una dirección, donde será posible encontrar datos que deben ser del tipo asociado al puntero. Para declarar una variable de tipo puntero basta con hacer preceder al tipo asociado con el símbolo "^". Por ejemplo, para declarar un puntero a un entero se podría hacer de la siguiente forma: "*n :^integer;*". La variable *n* no contiene ahora el dato de tipo entero, sino la dirección donde se puede encontrar dicho dato; así, si se trata de asignar el valor *-5* a la variable *n* se producirá un error. Para hacer referencia al contenido de la posición de memoria apuntada por *n* basta con hacer suceder al identificador del puntero con el símbolo "^", de tal forma que el incremento del contenido de la posición de memoria apuntada por *n* se realizaría de la siguiente forma: "*n^ :=n^+1 ;*". De todas formas, antes de poder asignar algún valor al contenido señalado por un puntero es necesario solicitar una porción de memoria adecuada para albergar los datos y que dicha dirección se almacene en la variable puntero. El procedimiento **new** permite reservar una zona de memoria del *heap*; la dirección de esta zona será el contenido de la variable puntero que se pasa como parámetro al procedimiento mientras que el tamaño de la zona de memoria se corresponderá con el tamaño del tipo asociado al puntero.

En la Figura 1 se puede observar la declaración de una variable de tipo puntero *punt_per* que tiene como tipo base el registro *persona*, al igual que cualquier otro tipo de variable no contiene inicialmente un valor válido. Mediante el procedimiento *new(punt_per)* se ha reservado memoria suficiente para albergar los datos de un registro, pero la variable *punt_per* no contiene los datos del registro sino la dirección donde éstos se encuentran, sin embargo, se puede acceder a los datos del registro haciendo suceder a la variable de tipo puntero el símbolo "^", que indica la entrada

FIGURA 1.

```
type
 persona = record
             nombre : string[15] ;
             direccion : string[70] ;
             tfno : string[10] ;
             edad : integer ;
           end  ;
var
 punt_per : ^persona ;
 ...
begin
 ...
 new(punt_per) ;
 punt_per^.nombre := 'Pepe' ;
 punt_per^.direccion := 'Avda. de America, 33' ;
 ....
end.
```

```
procedure inserta_lista_ordenada
(elem : char ; var l : lista) ;
var
 actual, adelante :lista ;
 aux : lista ;
begin
 if (l = nil)
  then
  begin
   new(aux) ;
   aux^.siguiente := nil ;
   aux^.elemento := elem ;
   l := aux ;
  end
  else
   if (l^elemento > elem)
     then
     begin
      new(aux) ;
      aux^.siguiente := l ;
      aux^.elemento := elem ;
      l := aux ;
     end ;
else
begin
 actual := l;
 adelante := l^.siguiente ;
 while (adelante <> nil) and
         (adelante^.elemento <= elem)
  do
   begin
    actual := adelante ;
    adelante := adelante^.siguiente ;
   end ;
 if (actual^.elemento <> elem)
   then
    begin
     new(aux) ;
     aux^.sig := adelante ;
     aux^.elemento := elem ;
     actual^.sig := aux ;
    end ;
 end ;
end ;
```

FIGURA 2.

al contenido de los datos de la dirección contenida en el puntero. De esta forma, *punt_per^* hace referencia a la totalidad del registro, mientras que *punt_per^.nombre* se refiere al campo *nombre* del registro. Como es lógico, cualquier dato accedido a través de un puntero puede ser utilizado en cualquier ambiente en que se pueda utilizar el tipo de datos resultante, por ejemplo, a *punt_per^.nombre* se le podría aplicar cualquier operación asociada a una cadena de caracteres.

El procedimiento ***dispose***, que toma como parámetro una variable de tipo puntero, permite liberar la zona de memoria referenciada por el puntero, quedando ésta disponible para usos posteriores. Una vez que se ha aplicado este procedimiento a una variable de tipo puntero cualquier intento de referenciar la zona apuntada por el puntero será considerada como un error.

Además de los procedimientos *new* y *dispose*, Pascal cuenta con la constante ***nil*** que puede ser asignada a cualquier variable de tipo puntero, independientemente del tipo de datos al que referencia. Como se verá posteriormente servirá para indicar que un puntero referencia a la zona de memoria "nula".

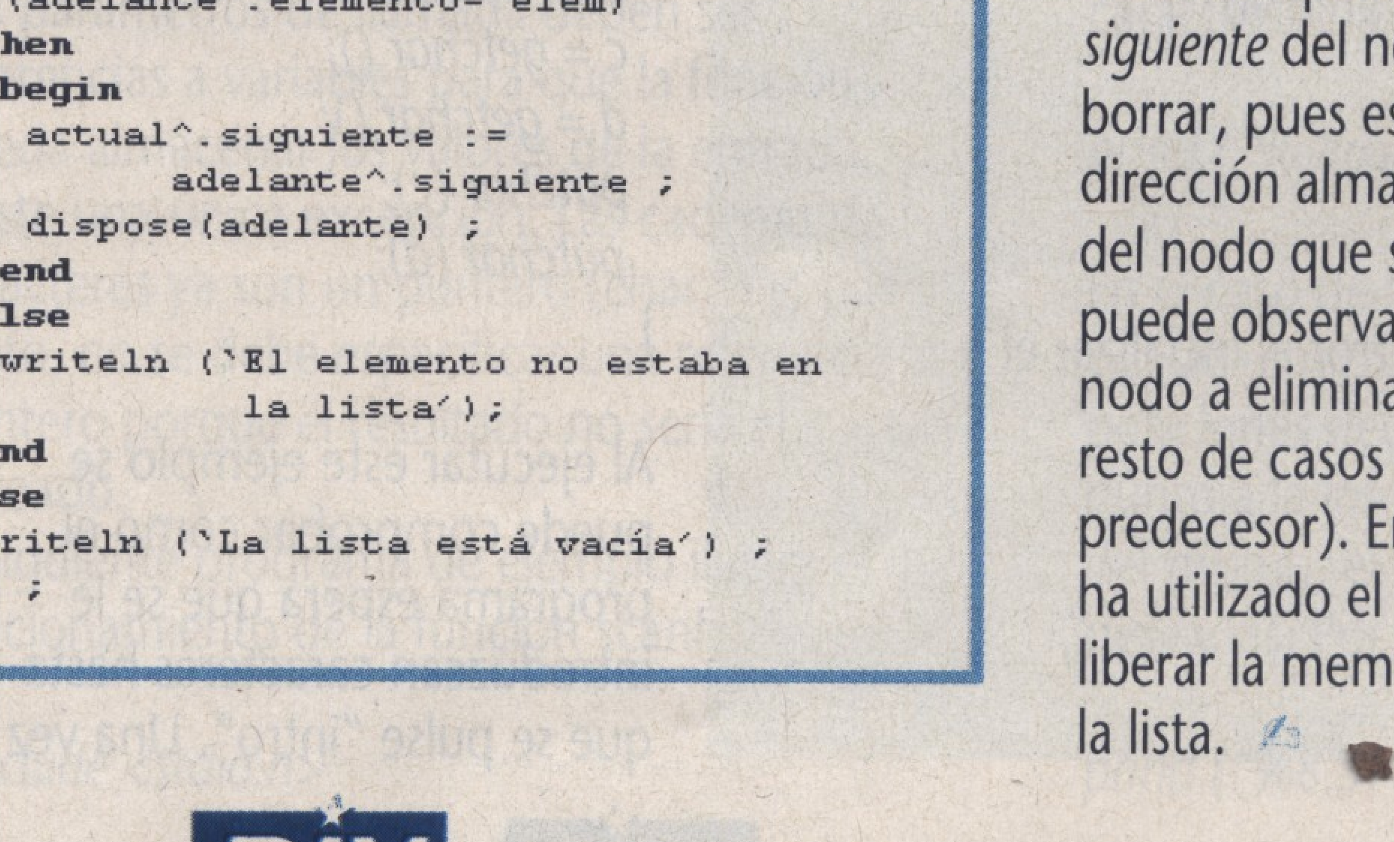
LISTAS ORDENADAS

Una de las aplicaciones más usuales otorgada a una lista consiste en mantener un conjunto de elementos ordenados, aunque no es posible aplicar más que la búsqueda secuencial; los procedimientos para el tratamiento de éstas ilustran adecuadamente las operaciones que normalmente se realizan con ellas y con los punteros.

En la Figura 2 puede observarse la inserción de un nuevo nodo en una lista ordenada. La primera acción consiste en comprobar si la lista se encuentra vacía (*l=nil*), si es así se procede a solicitar memoria para un nuevo nodo siendo el elemento siguiente a éste la constante ***nil*** (fin de la lista); el puntero que indica el primer nodo de la lista, *l*, apuntará al nodo insertado cuya dirección se encuentra almacenada en el puntero *aux*. Si la lista no se encontrara vacía, se distinguen dos casos. El primero consiste en que el elemento a insertar sea menor que todos los de la lista, puesto que ésta se encuentra ordenada; basta con comparar el elemento a insertar con el elemento almacenado en el primer nodo. La inserción en este caso se produce al principio de la lista siendo un proceso equivalente al anterior, salvo que el nodo insertado debe apuntar al que fue el primer nodo de la lista. En el segundo caso, se debe explorar la lista en busca de la posición adecuada para la inserción. Para realizar este proceso se utilizan dos punteros *actual* y *adelante*, en donde el segundo de ellos, *adelante*, siempre señalará al nodo sucesor apuntado por el puntero *actual*. Mediante un bucle se hace avanzar de forma simultánea a ambos punteros hasta que el puntero *adelante* señale un nodo que contenga un elemento mayor que el que se desea insertar, o bien, hasta que llegue al final de la lista. Una vez localizado el punto de inserción, se comprueba si el elemento no se encuentra en la lista; si es así, se solicita un nuevo nodo siendo éste el elemento que precederá al nodo apuntado por el puntero *adelante*, por tanto, en su campo *siguiente* debe almacenarse la dirección del puntero *adelante*. El nodo predecesor del nodo que se desea insertar está señalado por el puntero *actual*, el campo *siguiente* de este nodo deberá apuntar a la dirección del nodo a insertar. En este punto es posible comprender la necesidad de utilizar dos punteros para realizar el proceso de inserción.

En la figura 3 puede observarse el procedimiento para eliminar un nodo de la lista. Este sistema es similar al de inserción de una lista en lo que se refiere a la utilización de dos punteros, uno para indicar el nodo a eliminar y otro para apuntar al nodo predecesor de éste. El puntero *actual* es necesario para poder acceder al campo *siguiente* del nodo que precede al que se desea borrar, pues este campo pasará a contener la dirección almacenada en el campo *siguiente* del nodo que se va a eliminar. También aquí se puede observar la asimetría del caso en que el nodo a eliminar sea el primero de la lista con el resto de casos (el primer nodo no tiene predecesor). En este procedimiento también se ha utilizado el procedimiento ***dispose***, para liberar la memoria de los nodos eliminados de la lista.

```
procedure elimina_elemento
 (var l :lista ; elem : char) ;
var
 actual, adelante : lista ;
begin
 if (l <> nil)
   then
    if (l^.elemento = elem)
      then
       begin
        actual := l ;
        l := l^.siguiente ;
        dispose(actual) ;
       end
   else
    begin
     actual := l ;
     adelante := l^.siguiente ;
```



```
while (adelante <> nil) and
       (adelante^.elemento < elem)
    do
     begin
      actual := adelante ;
      adelante:=adelante^.siguiente ;
     end ;
if (adelante <> nil) and
   (adelante^.elemento= elem)
  then
   begin
    actual^.siguiente :=
          adelante^.siguiente ;
    dispose(adelante) ;
   end
  else
   writeln (`El elemento no estaba en
              la lista´);
  end
 else
  writeln (`La lista está vacía´) ;
end ;
```

FIGURA 3.

PROGRAMACIÓN EN C

Entrada y salida (I)

Es difícil concebir un programa útil que no utilice entrada y salida de información. En esta entrega de la entrada/salida en C se tratan la consola y los archivos de texto.

C proporciona numerosas funciones para la gestión de la entrada/salida de información. Existen dos grandes grupos. Por un lado se encuentran las funciones destinadas a trabajar con la consola, es decir, la pantalla y el teclado. Por otro lado están las funciones orientadas al trabajo con archivos.

CARACTERES

El tratamiento de la entrada y salida de caracteres por teclado y pantalla se reduce a dos sencillas funciones. Estos son sus prototipos, que se pueden encontrar en la biblioteca "stdio":

• **getchar:**

Prototipo: *int getchar (void);*

Esta es la función encargada de capturar un carácter desde teclado. Sin embargo, por compatibilidad con la primera versión de C para el sistema operativo UNIX, su funcionamiento no es exactamente el que cabía esperar. En lugar de leer la primera tecla pulsada y devolver el carácter asociado, espera a que se llene el buffer para leer el primer carácter de la entrada. Y para llenar el buffer de entrada es necesario pulsar "intro". Debido a este curioso comportamiento, la mayoría de compiladores incluyen funciones no estándar para tratar la situación de forma adecuada. Normalmente estas funciones reciben el nombre "getch" y "getche".

• **putchar:**

Prototipo: *int putchar (int);*

La misión de *putchar* es mostrar en pantalla el carácter pasado como parámetro. A diferencia de la función anterior, en esta ocasión el funcionamiento sí se ajusta a lo esperado.

El lector recordará del capítulo anterior que todas las funciones de la biblioteca estándar utilizan enteros para tratar con los caracteres. Estas dos funciones, como puede verse, no son una excepción y del mismo modo utilizan el byte de menor peso del entero para representar el carácter. El siguiente ejemplo sirve para ilustrar claramente el comportamiento de *getchar* y *putchar:*

```
#include <stdio.h>

void main () {
  char c, d;
  c = getchar ();
  d = getchar ();
  putchar (c);
  putchar (d);
}
```

Al ejecutar este ejemplo se puede comprobar cómo el programa espera que se le introduzcan caracteres hasta que se pulse "intro". Una vez hecho esto, muestra en pantalla los dos primeros caracteres tecleados.

LISTADO 1

```
#include <stdio.h>

#define ERROR -1
#define OK 0

int main () {
  FILE *fich;
  char c;
  if ((fich = fopen ("info.txt", "w")) != NULL) {
    printf ("\nIntroduzca una serie de caracteres.\n");
    printf ("No olvide pulsar \"intro\" al menos una vez.\n");
    printf ("Para finalizar teclee el caracter '@'\n");
    do {
      if ((c = getchar ()) != '@') {
        fputc (c, fich);
      }
    } while (c  != '@');
    if (fclose (fich)) {
      printf ("Error en el cierre del archivo\n");
      return (ERROR);
    }
  }
  else {
    printf ("Se ha producido un error al incializar el archivo\n");
    return (ERROR);
  }
  return (OK);
}
```

CADENAS

La entrada/salida de cadenas, como es lógico, es algo más complicada. En este caso también existen dos funciones básicas llamadas "gets" y "puts" cuyo prototipo se puede encontrar en el archivo de cabecera "stdio.h". Sin embargo, se pueden encontrar otras dos funciones sumamente útiles para realizar entrada y salida de cadenas con formato. Se trata de "printf" y de "scanf". La primera se ha utilizado en innumerables ocasiones a lo largo del curso, mientras que la segunda es completamente nueva y por ello será tratada en profundidad. A continuación se explican las tres funciones mencionadas anteriormente:

• **gets:**

Prototipo: *char *gets (char *cadena);*

La función gets captura una cadena introducida por teclado y la almacena en memoria a partir de la dirección de memoria apuntada por el puntero cadena. Se leen caracteres de la entrada hasta que se encuentra un retorno de carro. Sin embargo, el retorno de carro no se almacena en destino. En su lugar se almacena un '\0' que indica el fin de la cadena. Como suele ser habitual, C no hace ninguna comprobación acerca del destino. Por ello el programador debe asegurarse siempre que hay espacio suficiente en la zona de memoria apuntada por "cadena" para almacenar la entrada. En caso contrario, se podrían sobreescribir otros datos del programa y producirse un funcionamiento anómalo.

• **puts:**

Prototipo: *int puts (char *cadena);*

La función puts es la opuesta a gets. Su trabajo consiste en imprimir en pantalla el conjunto de caracteres apuntados por el parámetro "cadena". En caso de error, puts devuelve el valor EOF. Además, añade el carácter nueva línea al final de la cadena en la salida.

Un sencillo ejemplo del funcionamiento de estas dos funciones puede ser el siguiente:

```
#include <stdio.h>

void main () {
  char cadena[100];
  gets (cadena);
  puts (cadena);
}
```

Este pequeño programa se limita a leer una cadena desde teclado y posteriormente la muestra por pantalla. Se asume que la longitud del array cadena será más que suficiente para almacenar la entrada.

- **scanf:**

Prototipo: *int scanf (const char *fmt, ...);*

En el caso de necesitar interpretar una entrada con un formato determinado, scanf es la función apropiada. Al igual que printf, esta función utiliza una cadena de formato que utiliza unos códigos especiales para realizar su tarea. Estos códigos indican a scanf el tipo de datos que debe esperar de la entrada. La cadena de formato se interpreta de izquierda a derecha haciendo corresponder los códigos con los restantes argumentos pasados a la función, en el mismo orden. En la tabla 1 se especifican los códigos válidos. Son muy parecidos a los utilizados para printf (ver el capítulo referente a los arrays, las cadenas y las estructuras), salvo el "%[]". Sirve para definir un conjunto de caracteres aceptables en la entrada. Por ejemplo:

%[ilus]

El código anterior aceptaría una cadena como "luis". Sin embargo, no reconocería "luisa" porque la "a" no se encuentra en el conjunto. En este último caso, scanf reconocería sólo "luis" y continuaría interpretando la entrada a partir de la "a". Por otro lado, también es posible especificar los caracteres que no deben aparecer en la entrada. Para ello se especifica como primer carácter del conjunto el "^". Si en el ejemplo anterior se hubiera optado por el siguiente código:

%[^ilus]

Ante la cadena "abcluis", scanf reconocería la cadena "abc", ignoraría el carácter "l" (porque está en el conjunto) y continuaría el reconocimiento a partir de la "u" siguiendo las directrices del resto de la cadena de formato. También existe la posibilidad de indicar un rango:

%[a-n]

Con este código se reconoce cualquier cadena que contenga cualquiera de los caracteres comprendidos entre la "a" y la "n". Respecto a los conjuntos sólo queda decir que únicamente se pueden hacer corresponder con cadenas de caracteres, es decir, con un puntero a carácter. No se pueden utilizar para reconocer ningún otro tipo de dato.

Además, en la cadena de formato se pueden especificar espacios en blanco. Un espacio en blanco indica a scanf que ignore espacios en blanco, un salto de línea o una tabulación. Cualquier otro carácter distinto del espacio indica a scanf que ignore dicho carácter. En caso de no aparecer el carácter esperado, la función se detiene. Si en algún caso es necesario interpretar un espacio como tal, basta con reconocerlo como si fuera un carácter ("%c") y será asimilado sin problemas. También se puede ignorar un elemento de la entrada. Aprovechando el ejemplo anterior, ante la entrada "luisas" y el código de formato "%[ilus]%*c%c", scanf reconocería lo siguiente:

Cadena: "luis"
Carácter: "s"

Como puede verse, el carácter "a" desaparece de la entrada. Esto es aplicable a cualquier tipo de dato admitido por la función además de los caracteres individuales.

Como es de suponer, en la mayoría de los casos los elementos de la entrada deben estar separados mediante cualquier separador, incluido el espacio en blanco. Exceptuando casos como el especificado anteriormente con los conjuntos, la función sería incapaz de distinguir el principio y el final de cada uno de los elementos. Por ejemplo, ante la entrada "hola1234" scanf reconoce una sola cadena, en lugar de una cadena ("hola") y un entero ("1234"), que parece lo más lógico. Para resolver un caso como éste se puede especificar la longitud en caracteres del dato esperado. Un ejemplo:

%4s

Mediante este código se reconocería una cadena de cuatro caracteres de longitud.

Los parámetros de llamada deben ser referencias a variables para que la función pueda almacenar los valores de la entrada. Existe una única excepción. Las cadenas de caracteres ya son un puntero (char *) y, por tanto, no se debe especificar una referencia al puntero porque el resultado no sería el deseado.

El siguiente programa de ejemplo ilustra el funcionamiento de la función scanf:

```
#include <stdio.h>

void main () {
  char cadena[15], caracter;
  int entero;
  printf ("Introduzca los datos con el siguiente formato:\n");
  printf ("Cadena Entero, Carácter\n");
  scanf ("%s %d, %c", cadena, &entero, &caracter);
  printf ("\nLos datos introducidos han sido:\n");
  printf ("Cadena: %s\n", cadena);
  printf ("Entero: %d\n", entero);
  printf ("Carácter: %c\n", caracter);
}
```

La función scanf ofrece multitud de posibilidades. Por ello, la única manera de dominar su uso es mediante la práctica. Se recomienda al lector que realice numerosas pruebas antes de continuar.

CONSIDERACIONES SOBRE PRINTF

Hace dos entregas fue necesario tratar por encima la función printf. Sin embargo no se trataron todos los aspectos relacionados con la misma porque el objetivo en aquel momento era otro muy distinto. Ha llegado el momento de completar la explicación sobre su funcionamiento.

Al igual que scanf, esta función admite que se especifique la longitud de la entrada. En este caso se indica la longitud mínima de la salida en lugar de la longitud máxima de la entrada. El formato es el mismo:

%<longitud><carácter de formato>

Si el dato no cubre el mínimo espacio especificado la salida se rellena con espacios en blanco o ceros. En este último caso, se debe colocar un cero delante de la longitud mínima. Por ejemplo, la siguiente sentencia mostraría un entero con una longitud mínima de 8 caracteres, justificado a la derecha y rellenando con ceros:

```
printf ("%08d\n", i);
```

Si se desea justificar la salida a la izquierda se debe añadir un signo menos detrás del carácter '%' de la cadena de formato:

```
printf ("%-8d\n", i);
```

En el caso de los números de coma flotante se puede especificar también el número de posiciones decimales. Para ello se añade un punto a la longitud mínima y, a continuación del mismo, el número de posiciones decimales deseadas. Por ejemplo:

```
printf ("%8.5f\n", f);
```



Operaciones sobre bits

Se puede decir que saber usar las instrucciones de manejo de bits es fundamental para todo aquel que tenga la intención de crear algoritmos bajo ensamblador más o menos complejos.

En lo que va del presente curso, se ha explicado ya un gran número de conceptos relacionados con el funcionamiento interno del PC y la CPU, así como un número ya considerable de instrucciones. Ahora pasaremos a profundizar en otra de las áreas básicas de la programación en ensamblador, que es todo lo referente al manejo de bits en general.

PARA QUÉ SIRVEN

Básicamente, podemos decir que la potencia de un determinado microprocesador viene dada, aparte de por los propios MIPS (*Millones de Instrucciones Por Segundo*) que puede desarrollar, por la variedad de instrucciones que disponga en su set para operar con bits (entre otras, claro).
Un ejemplo de ello es la familia de chips POWER PC, que (aparte de ser RISC) gracias a que poseen una completa librería de dichas instrucciones, son aptos para poder ser usados como plataforma desde la cual poder emular otras máquinas (por ejemplo, Pentium), ya que la emulación se basa mucho en el control de bytes y bits individuales y en caso de tener pocas instrucciones destinadas a tales fines, el código tiene que ejecutar muchas más instrucciones para realizar las mismas funciones, con el consecuente aumento de ciclos de reloj necesarios, lo cual se traduce en un menor rendimiento del programa.
Al igual que este ejemplo mencionado, podemos decir que estas instrucciones son fundamentales en general en todo tipo de algoritmos que podamos realizar y además su utilidad aumenta de forma directamente proporcional a la complejidad de los mismos.

CLASIFICACIÓN DE LAS INSTRUCCIONES

Las instrucciones de bits que hemos dicho que tenemos se dividen, a su vez, en tres clases según

```
INSTRUCCIONES LÓGICAS BÁSICAS

EJEMPLO OPERACIÓN <OR>          OPERACIÓN <AND>
 0 1 0 1 0 1 0 1                 0 1 0 1 0 1 0 1
 0 1 0 0 1 1 0 1                 0 1 0 0 1 1 0 1
 ---------------                 ---------------
 0 1 0 1 1 1 0 1                 0 1 0 0 0 1 0 1
bit 7 -resultado- bit 0         bit 7          bit 0

OPERACIÓN <XOR>                 OPERACIÓN <NOT>
 0 1 0 1 0 1 0 1                 0 1 0 0 1 1 0 1
 0 1 0 0 1 1 0 1                 ---------------
 ---------------                 1 0 1 1 0 0 1 0
 0 0 0 1 1 0 0 0                bit 7          bit 0
bit 7             bit 0
```

FOTOGRAFIA 1: INSTRUCCIONES DE MANEJO DE BITS.

su utilidad. Así, disponemos de las de desplazamiento (englobando en ellas tanto lógicas como aritméticas), las de rotación simple y las de rotación con acarreo, aparte, cómo no, de algunas operaciones lógicas básicas que están presentes en todos los micros, al igual que otras estándar como son las de suma y resta por ejemplo.
La diferencia entre las operaciones lógicas y aritméticas es muy simple de comprender: mientras las aritméticas no alteran el signo del elemento sobre el cual operan, las lógicas no lo tienen en cuenta.
Si las mencionamos todas de forma clasificada, tenemos como operaciones básicas cinco instrucciones: AND, NOT, OR, TEST y XOR.
Como instrucciones de desplazamiento de bits, tenemos dos subclases: por un lado tenemos SHL y SHR como lógicas y las instrucciones SAL y SAR como aritméticas.
Por último, tan sólo quedan las llamadas de rotación, que también se dividen en dos subclases: las simples y las de acarreo. En la primera subclase disponemos de ROL y ROR y en las de rotación con acarreo, de RCL y RCR.
A continuación vamos a detallar casi todas para que el lector pueda entenderlas perfectamente y pueda usarlas en la práctica:

INSTRUCCIONES LÓGICAS BÁSICAS

AND destino,fuente:
Esta instrucción realiza la operación *<y lógica>* a nivel de bits entre los dos operandos. Los dos parámetros pueden ser tanto dos palabras (*words*) como dos bytes. La operación AND consiste básicamente en que los bits del primer operador se comparan con los correspondientes del otro que ocupan la misma posición y se ponen en destino a 1 sólo aquellos casos en que ambos bits poseían el valor 1. En la fotografía 1 tenemos un esquema del funcionamiento.
Lógica:

```
{
Destino= Destino "AND" fuente;
CF=0;
OF=0;
}
```

OR Fuente, Destino:
Esta instrucción, cuyas siglas proceden del inglés *<logical inclusive OR>*, realiza la llamada operación *o lógico inclusivo* a nivel de bits. Esta operación consiste en que cada uno de los bits de los operadores se comparan y se almacena 0 en destino en todos los casos en que ambos sean 0 y en caso de que uno o ambos posean el valor 1. Ver fotografía 1.
Lógica:

```
{
Destino = Destino <or> Fuente;
CF= 0;
OF=0;
}
```

NOT Destino:
Not, que es una simplificación de la operación *<no lógico>* (*logical NOT*), simplemente cambia el estado de todos los bits del operador indicado. Así pues, todos los bits que estuviesen inicialmente a 1, pasarán a tener el valor 0 y viceversa.
A esta operación se la conoce también con el nombre de *<complemento a uno>*. Para ver un ejemplo , mirar fotografía 1.
Lógica:

```
{
Si (Destino== byte)
  { Destino = FFh - Destino; }
Si (Destino== word)
  { Destino = FFFFFh - Destino; }
}
```

TEST Destino,Fuente:
Esta instrucción realiza la comparación lógica a nivel de bits (*Y lógico*), pero sin almacenar el resultado en destino. Por lo tanto, excepto este último dato, tenemos que realizar la misma operación que en la instrucción AND.
Lógica:

```
{
Destino <y lógico> Fuente;
Actualización banderas;
CF=0;
OF=0;
}
```

XOR Destino,Fuente:

Esta instrucción realiza la operación <o lógico exclusivo> y proviene del inglés <logical eXclusive OR>. Su función es la de comparar los bits de los dos operandos y la de almacenar en destino un 0 siempre que ambos bits sean ceros o unos y un 1 sólo si son diferentes (0/1 o 1/0). Ver Foto 1.

Lógica:

```
{
Destino= Destino <xor> Fuente;
CF=0;
OF=0;
}
```

DE DESPLAZAMIENTO LÓGICO

SHL Destino,Contador:

Esta instrucción, que proviene del inglés <SHift logical Left>, realiza la operación <Desplazamiento lógico a la izquierda>.

Su función es la de desplazar los bits de destino tantas posiciones hacia la izquierda como esté indicado en *Contador*.

Los bits nuevos que van entrando por la derecha poseen valor 0.

Otro dato importante, y que se aplica en todas las instrucciones de desplazamiento y rotación que se explicarán, es que *Contador* tiene que ser por fuerza CL si se ha de indicar un valor de desplazamiento mayor a 1. En caso de que queramos indicar un desplazamiento (o rotación) unidad, tenemos la posibilidad de poder elegir entre si queremos indicarlo como un valor inmediato o con CL.

Lógica:

```
{
Temp=Contador;
Mientras(Temp!=0)
  {CF=Bit superior Destino;
   Destino= Destino*2;
   Temp—;
   Si(Contador==1)
      Si(Ultimo bit destino!=CF) OF=1;
      Sino: OF=0;
   Sino: OF=?;
  }
}
```

SHR Destino,Contador:

Proviene del inglés <SHift logical Right> y realiza la operación inversa a la anterior: desplaza *Destino* tantos bits hacia la derecha como esté indicado en *Contador*. Los bits que van quedando vacíos por la izquierda se van llenando con ceros.

Su lógica es la siguiente:

```
{
Temp=Contador;
Mientras(Temp!=0)
  {CF=Bit inferior Destino;
   Destino= Destino/2;
   Temp—;
   Si(Contador==1)
      Si(Ultimo bit destino!=Bit anterior) OF=1;
      Sino: OF=0;
   Sino: OF=?;
  }
}
```

DE DESPLAZAMIENTO ARITMÉTICO

SAL Destino,Contador:

Esta instrucción sería la versión aritmética de SHL y existe como instrucción válida de ensamblador, pero debido a una coincidencia de funcionamiento de tipo lógico, el resultado que produce es idéntico al generado por SHL y por esta razón físicamente es la misma instrucción máquina con otro mnemotécnico ensamblador. Aclarado esto, es fácil de entender que no se va a detallar más esta instrucción ya que la explicación sería exactamente la misma que la de SHL.

SAR Destino,Contador:

Estas siglas, que provienen del inglés <Shift Arithmetic Right> (*Desplazamiento Aritmético a la Derecha*), desplazan los bits de *Destino* tantos lugares hacia la derecha como se indique en CL, al igual que hacía SHR, pero con la diferencia de que en cada desplazamiento, el nuevo bit que entra por la izquierda contiene el valor del bit que ocupaba esa misma posición antes, con lo que tenemos que no altera el signo de *Destino* (que como sabemos, viene determinado por el último bit).

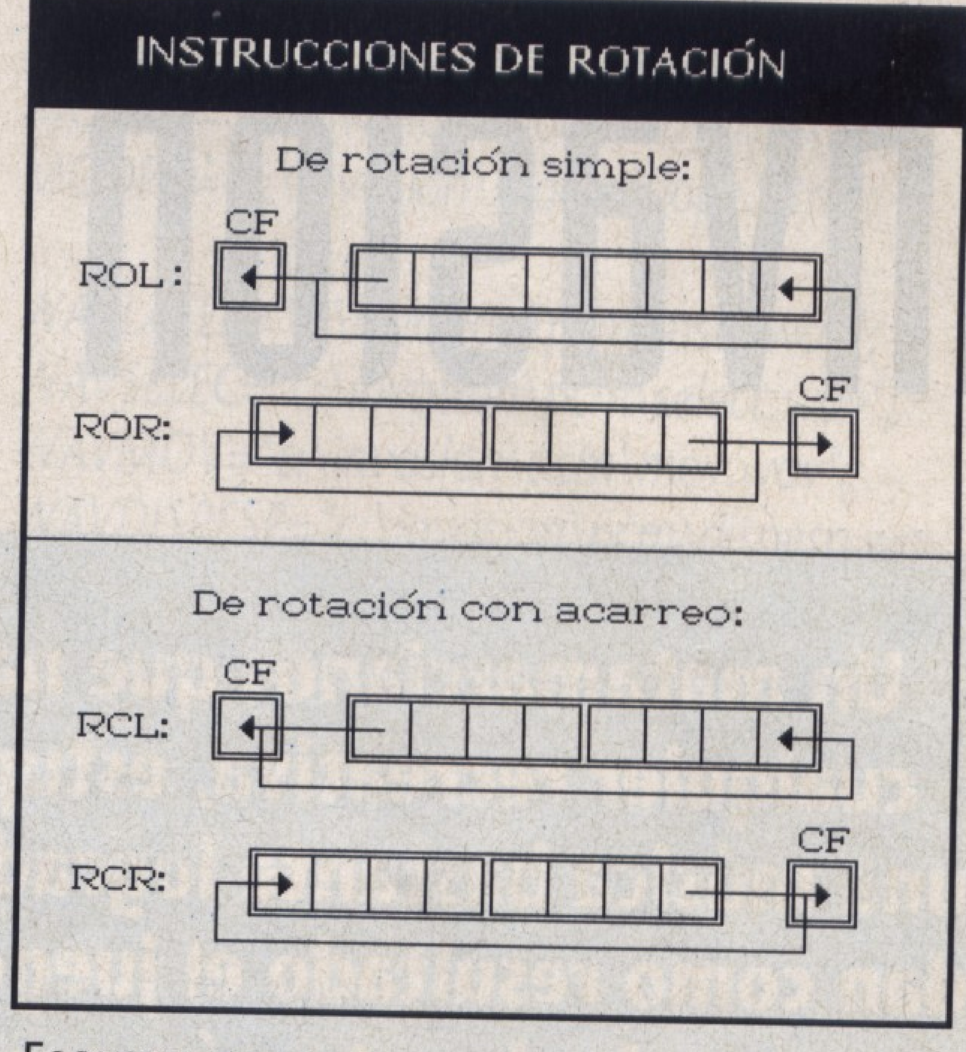
Lógica:

```
{
Temp=Contador;
Mientras(Temp!=0)
  {CF=Bit inferior Destino;
   Destino= Destino/2;
   Temp—;
   Si(Contador==1)
      Si(Ultimo bit destino!=Bit anterior) OF=1;
      Sino: OF=0;
   Sino: OF=?;
  }
}
```

DE ROTACIÓN SIMPLE

ROL Destino,Contador:

ROL son las siglas del inglés <ROtate Left>, lo cual se traduce simple y llanamente como <Rotación hacia la Izquierda>. Su funcionamiento es muy simple y muy similar al de SHL/SAL: realiza un desplazamiento de los bits de *Destino* el número *Contador* veces como dichas instrucciones, pero además el bit que va saliendo por la izquierda en cada ciclo de rotación no se pierde y se escribe tanto en CF como en el sitio que queda vacío tras el desplazamiento, que es el bit de más a la derecha del elemento *Destino*. Al final, con todo ello, gracias a esta forma de funcionar, tenemos que los bits no se pierden nunca, sino que tan sólo rotan hacia la izquierda como si de un circuito cerrado de bits se tratara. Por ejemplo, si rotamos 16 veces AX, el resultado quedaría inalterado, ya que se realizaría toda la vuelta completa y los bits volverían a ocupar la posición original en la que estaban (como decir que el sitio más lejos al que puedes ir es donde estás).



ESQUEMA DE LAS INSTRUCCIONES DE ROTACION.

Lógica:

```
{
Temp=Contador;
Mientras(Temp!=0)
  {CF=Bit Superior Destino;
   Destino= Destino*2 +CF;
   Temp—;
   Si(Contador==1)
      Si(bit superior destino!=CF) OF=1;
      Sino: OF=0;
   Sino: OF=?;
  }
}
```

ROR Destino,Contador:

ROR proviene, claro está, de <Rotate Right> y realiza la misma operación que ROL pero en sentido contrario: hacia la derecha, colocando por lo tanto, los bits que van saliéndose en cada desplazamiento (cuyo número son los indicados en *Contador*) por la derecha en el primer bit de la izquierda de *Destino* y en CF. Tenemos por lo tanto, que es simplemente una rotación cerrada de bits.

Lógica:

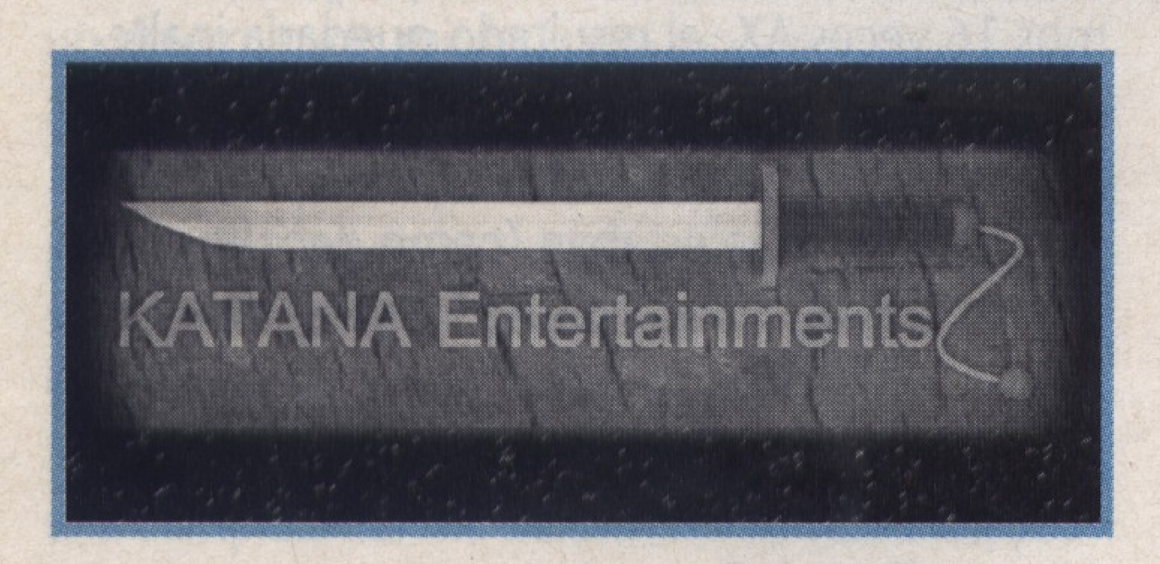
```
{
Temp=Contador;
Mientras(Temp!=0)
  {CF=Bit inferior Destino;
   Destino= Destino/2;
   Bit superior Destino=CF
   Temp—;
   Si(Contador==1)
      Si(bit superior destino!=bit anterior) OF=1;
      Sino: OF=0;
   Sino: OF=?;
  }
}
```

JUEGOS GANADORES: 1º

Autores: Juan C. Martínez
Ferrán M. Espinosa

Invasion

Un matamarcianos que nada tiene que envidiar a los de cualquier recreativa, estupendos vídeos entre misiones y una música bastante lograda; las tres características unidas dan como resultado el juego que se ha alzado con el primer premio de este número. Enhorabuena a los autores.



"Nos encontramos en el año 2537, una era en que la humanidad ha logrado, finalmente, expandirse a otros planetas y galaxias. Desgraciadamente, debido a su violenta naturaleza, esto solo ha dado lugar a nuevos conflictos bélicos con otros planetas y razas.

"Una pequeña resistencia de Marte ha decidido eliminar la Tierra, sede central de los humanos), para poder apaciguar esta era de destrucción y poder disfrutar de un poco de paz y tranquilidad.

"Antes de poder entrar al planeta, tienes que dirigirte a la luna (su satélite natural) para destruir su principal vía de comunicaciones con otras colonias, y así evitar que lleguen refuerzos. Dada la importancia de este satélite, hay un fuerte desplegamiento de tropas de defensa a su alrededor. Debes superar esta barrera y llegar a la luna.

"Fin de la transmisión.

Así empieza la intro del juego y, en seguida, nos colocamos en nuestro papel de marciano cabreado y nos acercamos a la Luna con ánimos hostiles contra los humanos. En una de las dos naves disponibles, tendremos que acabar con toda la resistencia, primero aérea, después terrestre y aérea y, para acabar, enfrentarnos a la terrible nave de final de fase. Podremos coger varios items en nuestro camino que nos proveerán de armas varias, escudos, bombas y rayos paralizantes, y todo mientras ponemos a prueba, una vez más, nuestros entrenados reflejos. Una calidad semejante a una recreativa, si no lo creéis cargar el juego en vuestro disco duro.

Bueno, ahora vamos a oír lo que nos dicen los autores de *Invasion:*

Hola !!!! Saludos redacción de Divmanía. Somos unos de los muchos que os enviamos un juego (bien suponemos que ya os lo imaginais por el CD), nuestro primer juego. Somos dos compañeros de instituto, yo, Juan C. Martínez (17 años, el Diseñador y Grafista), y Ferrán Miro Espinosa (16 años, el Programador). Este es nuestro primer trabajo juntos (y separados), es decir que es el primer juego que hacemos. Ferrán había empezado con *Div* hace mas o menos un año y yo con el *3DStudio2.5*, hace unos 6 meses.

Este juego (a partir de ahora lo llamaremos Invasion), lo empezamos hace mas o menos 3 meses, y al principio sólo era para ver como funcionaban las herramientas que hemos utilizado. Nos hemos dado cuenta que, tanto el Div como el 3DS, tenían muchas posibilidades. En el juego, al principio, no teníamos nada planeado, es decir estabamos desorganizados y lo hacíamos sobre la marcha (de ahí que hayamos tardado tanto en hacerlo), pero poco a poco empezamos ha diseñar los niveles, las trayectorias de los enemigos... fue entonces cuando todo se aceleró. De hecho hicimos mucho más juego en las últimas tres semanas que en todo el otro tiempo.

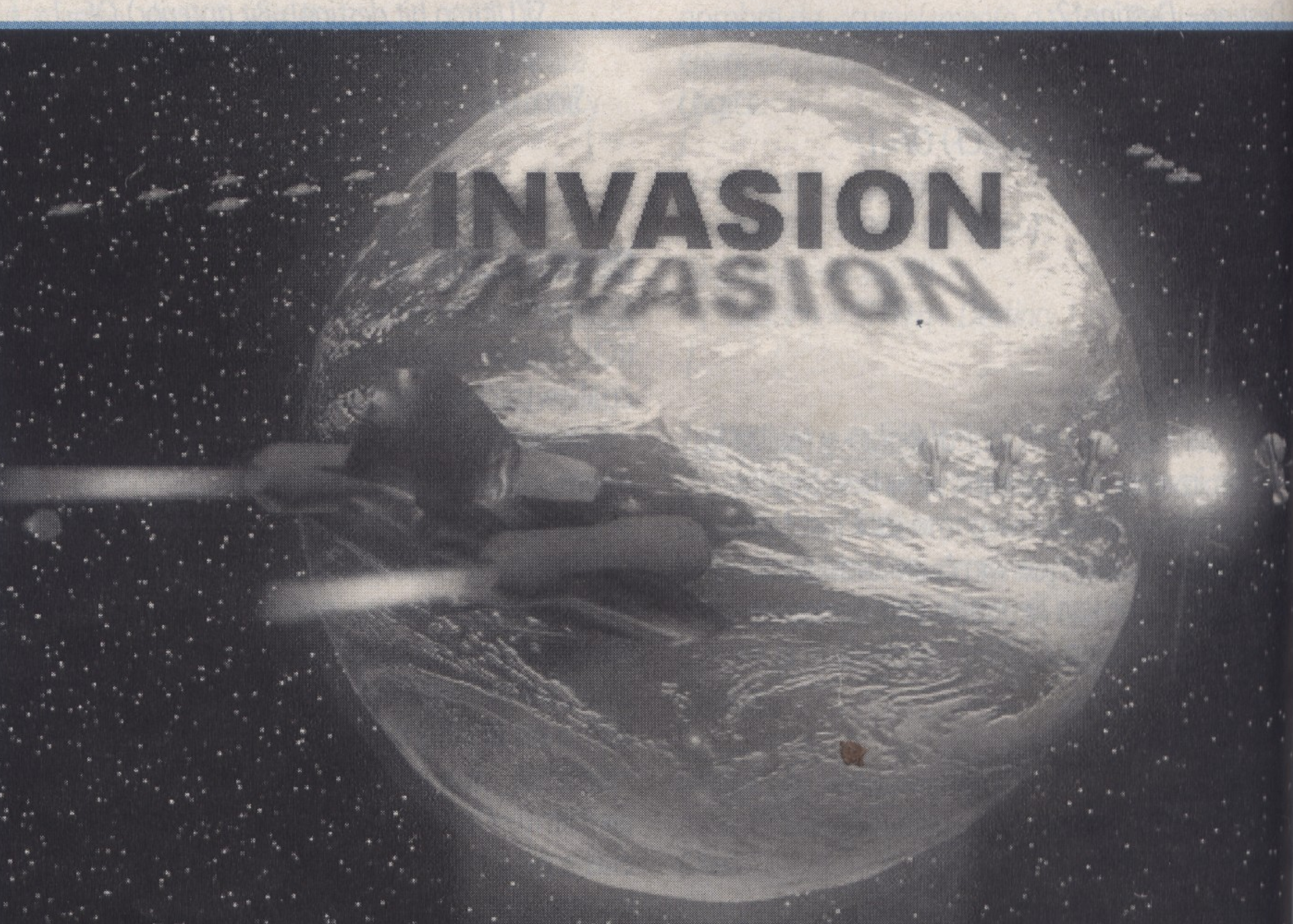
Ahora pasamos a algunas explicaciones sobre el juego ya sea en el apartado de programación como en el gráfico.

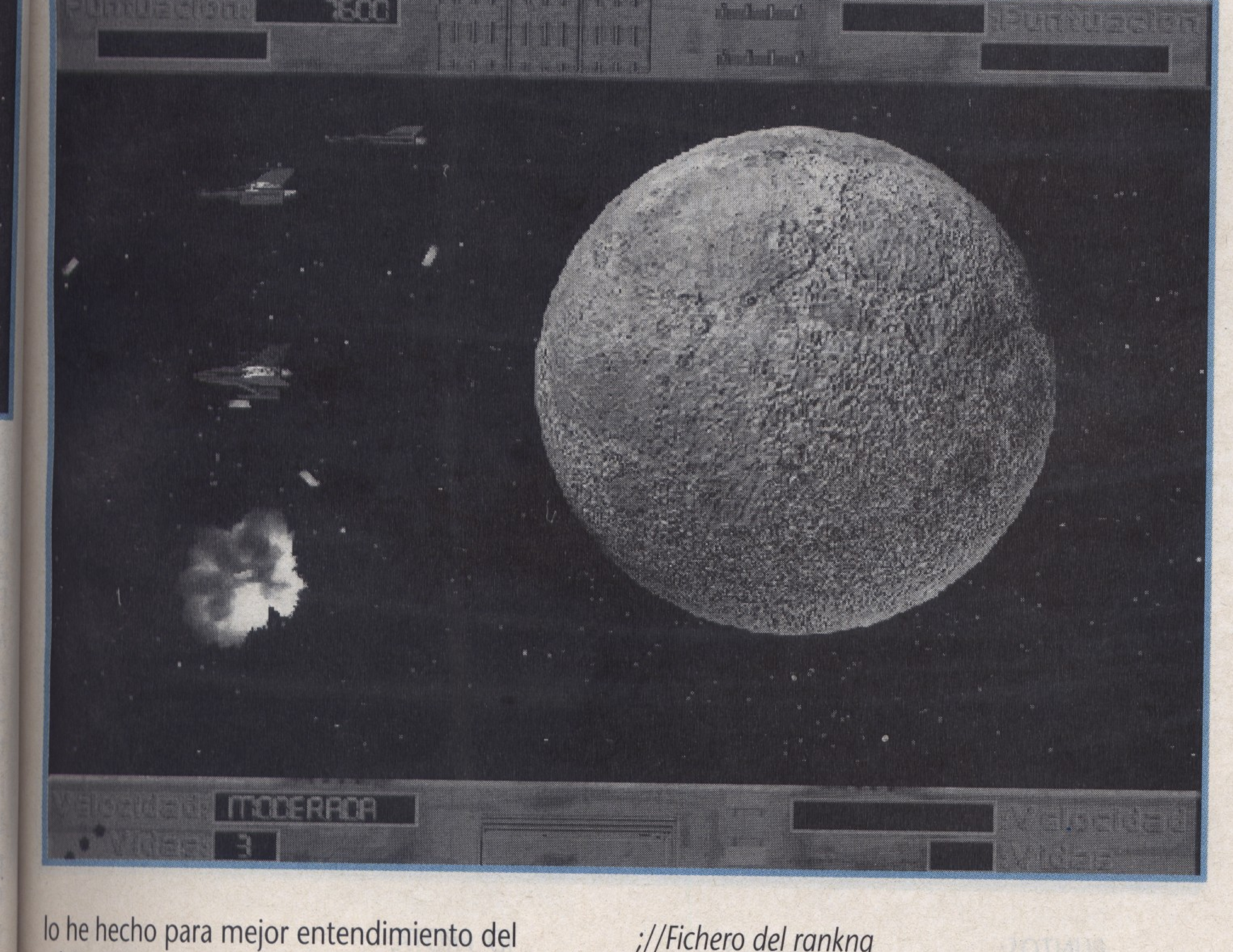
Nos encontramos en el año 2537, una era en que la humanidad ha logrado,finalmente, expandirse a otros planetas y galaxias.
Desgraciadamente, debido a su violenta naturaleza, esto solo ha dado lugar a nuevos conflictos bélicos con otros planetas y razas.

Una pequeña resisténcia de Marte, ha decidido eliminar la Tierra (sede central de los humanos) para poder apacigüar esta era de destrucción y poder disfrutar de un poco de paz y tranquilidad.

PROGRAMACIÓN Y GRÁFICOS

El juego ha sido programado en un Pentium 100 con 32 megas de Ram, lo que me ha hecho esmerarme a la hora de usar los recursos para evitar relentizaciones (aunque hay algunas). El código no esta muy optimizado, ya que hay variables que se podían haber substituido por otras ya predefinidas, pero no





lo he hecho para mejor entendimiento del código. Encontraréis que la rutina de las trayectorias de los enemigos es parecida a la del blastemup. Esto es debido a que probé otras maneras de hacerlo (ej. dar los puntos extremos y que calculara él el incremento) pero todas eran demasiado lentas. El cambio de gráfico de los enemigos cuando son dañados lo podría haber hecho con una conversión de paleta en vez de copiarlos todos otra vez, pero cuando lo hice aún no sabia usar las rutinas de la paleta (ni me las había mirado).
En cuanto a gráficos, solo comentar que han sido realizados con *3DStudio 2.5* y retocados (algunos) con *Photoshop*.

CÓDIGO

```
//Empezado el 18/11/1999 terminado el
9/3/2000

COMPILER_OPTIONS _extended_conditions,
//Para trabajar como en DIV1
        _max_process=200;

PROGRAM INVASION_DEMO;

CONST

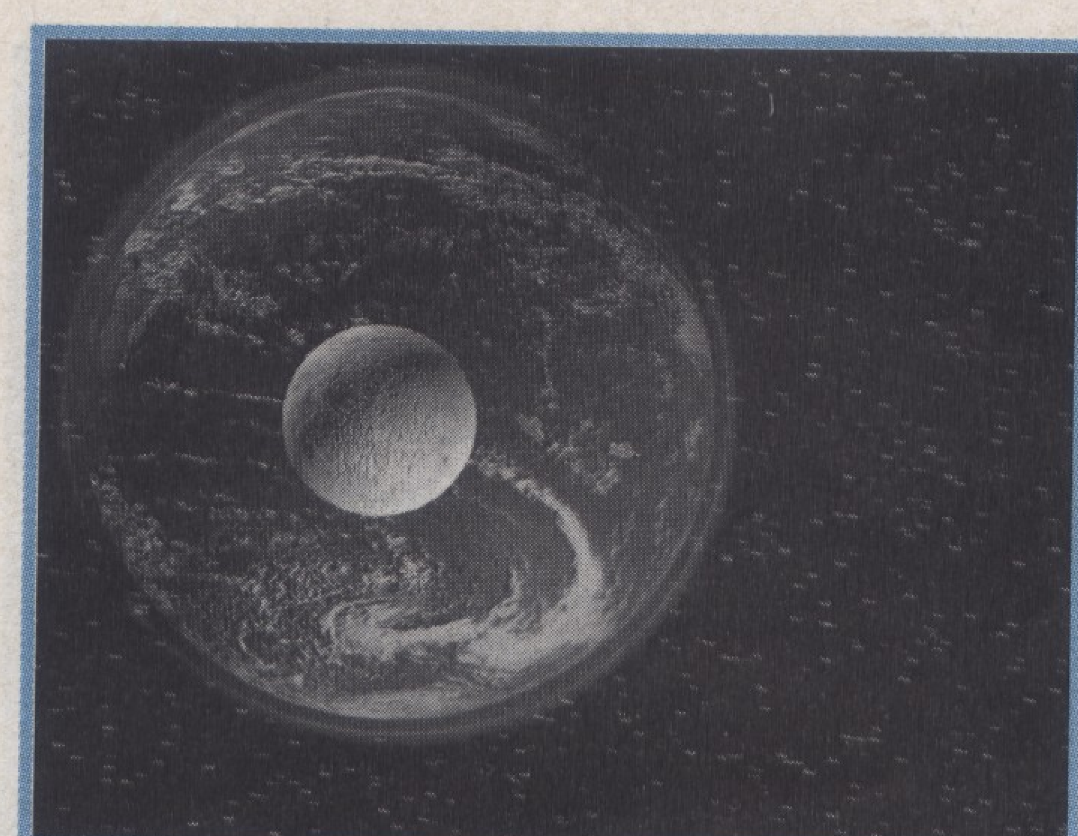
F_CONTROLES="c:\invasion\dat\controles.dat";/
/Fichero de la config. de controles

F_PUNTUACIONES="c:\invasion\dat\puntos.dat"
;//Fichero del rankng

  FPGMENUS="c:\invasion\fpg\menus.fpg";
  FPGN1="c:\invasion\fpg\n1.fpg";
  FPGN2="c:\invasion\fpg\n2.fpg";
  FPGN3="c:\invasion\fpg\n3.fpg";
  FPGPROTA1="c:\invasion\fpg\prota1.fpg";
  FPGPROTA2="c:\invasion\fpg\prota2.fpg";
  FPGEXPLO="c:\invasion\fpg\explosiones.fpg";
  FPGNAVES="c:\invasion\fpg\naves.fpg";
  FPGINICIO="c:\invasion\fpg\inicio.fpg";
  FPGENE1="c:\invasion\fpg\ene1.fpg";
  FPGENE2="c:\invasion\fpg\ene2.fpg";
  FPGENE3="c:\invasion\fpg\ene3.fpg";
  FPGENE5="c:\invasion\fpg\ene5.fpg";
  FPGBONUS="c:\invasion\fpg\bonus.fpg";
  FPGINFORMES="c:\invasion\fpg\info.fpg";

  FNT="c:\invasion\fnt\puntos.fnt";
  FNT1="c:\invasion\fnt\info.fnt";

  MODINICIO="c:\invasion\mod\inicio.xm";
  MODMENU="c:\invasion\mod\lenta.s3m";
```



```
  MODP1="c:\invasion\mod\uc-pajar.xm";
  MODP2="c:\invasion\mod\a0d_crpl.xm";
  MODP3="c:\invasion\mod\ciclope.s3m";

  WAVLASER="c:\invasion\pcm\laser.wav";
  WAVSELEC="c:\invasion\pcm\selecc.wav";
  WAVMOV="c:\invasion\pcm\mov.wav";
  WAVDISPESP="c:\invasion\pcm\dispesp.wav";

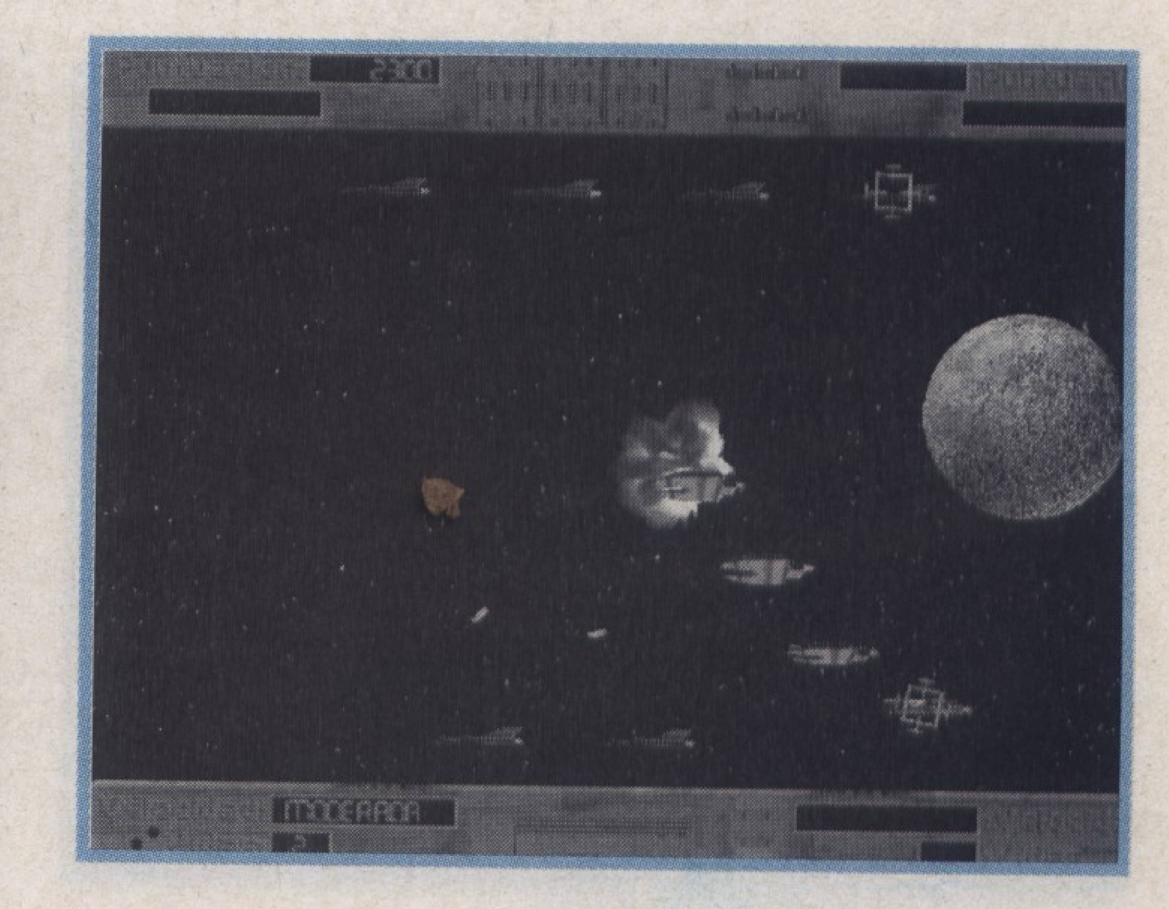
WAVEXPLOENE="c:\invasion\pcm\eploene.wav";
  WAVMIRILLA="c:\invasion\pcm\mirilla.wav";
  WAVBOMB="c:\invasion\pcm\bomb.wav";
  WAVBONUS="c:\invasion\pcm\bonus.wav";
  WAVALARMA="c:\invasion\pcm\alarma.wav";
  WAVLETRA="c:\invasion\pcm\letra.wav";
  WAVCONGE="c:\invasion\pcm\conge.wav";

  FLIPRESENT="c:\invasion\fli\pres2.fli";
  FLIBOSS1="c:\invasion\fli\enemic.fli";
  FLIKATANA="c:\invasion\fli\katana.fli";
  FLINIV1="c:\invasion\fli\niv1.fli";
  FLINIV2="c:\invasion\fli\niv2.fli";

GLOBAL

// La direccion de las fuentes

  ID_FUEGO[2];        //Aqui se guardan el ID de
los reactores de los protas
  N_JUGADORES;        //Numero de jugadores
que juegan
  SELECCIONANDO;      //Para saber cuando se
selecciona una opcion del menu
  ESC_PULSADO;        //Para cuando se pulse el
escape en un menu
  P_PULSADA;          //Para controlar la pausa
  TECLA_ASIGNADA;     //Guarda la tecla que
pulsamos cuando configuramos el teclado
  BOTON_ASIGNADO;     //Guarda la tecla que
pulsamos cuando configuramos el joystick
  MUSICA;             //Nos indica si la musica esta
activada o no(0=si,1=no)
  N_PANTALLA;         //Guarda la pantalla en
que estamos actualmente
  T_BARRA;            //ID del texto inferior de los
menus
  ID_TEXTO[15];       //ID de los textos de los
controles
```



```
DIFICULTAD;        //Dificultad seleccionada
VEL1;              //Velocidad del primer plano
del scroll

FMENUS;            //ID fpg
FN1;               //ID fpg
FN2;               //ID fpg
FN3;               //ID fpg
FPROTA1;           //ID fpg
FPROTA2;           //ID fpg
FEXPLO;            //ID fpg
FNAVES;            //ID fpg
FENE1;             //ID fpg
FENE2;             //ID fpg
FENE3;             //ID fpg
FENE5;             //ID fpg
FBONUS;            //ID fpg

FNT_1;             //FUENTE

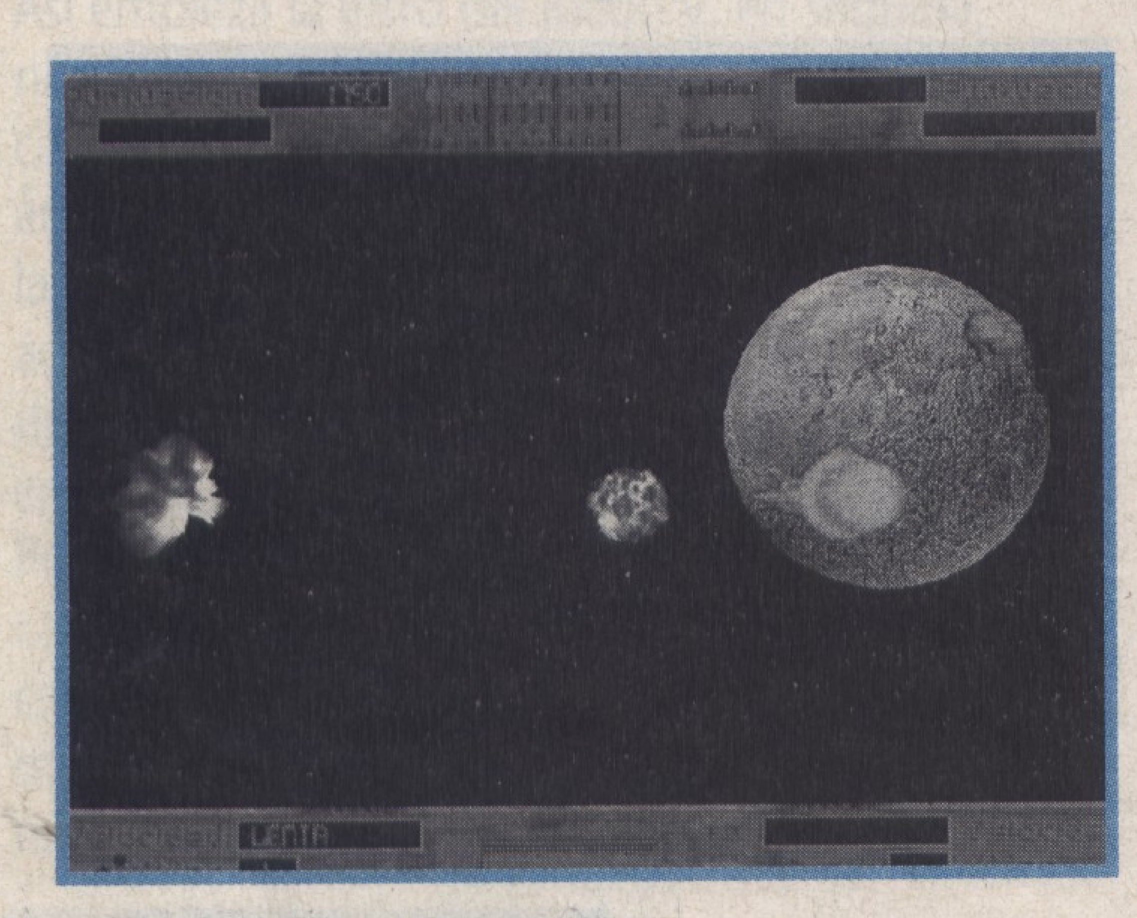
SLASER;            //SONIDO
SSELEC;            //SONIDO
SMOV;              //SONIDO
SDISPESP;          //SONIDO
SEXPLOENE;         //SONIDO
SMIRILLA;          //SONIDO
SBOMB;             //SONIDO
SBONUS;            //SONIDO
SALARMA;           //SONIDO
SCONGE;            //SONIDO

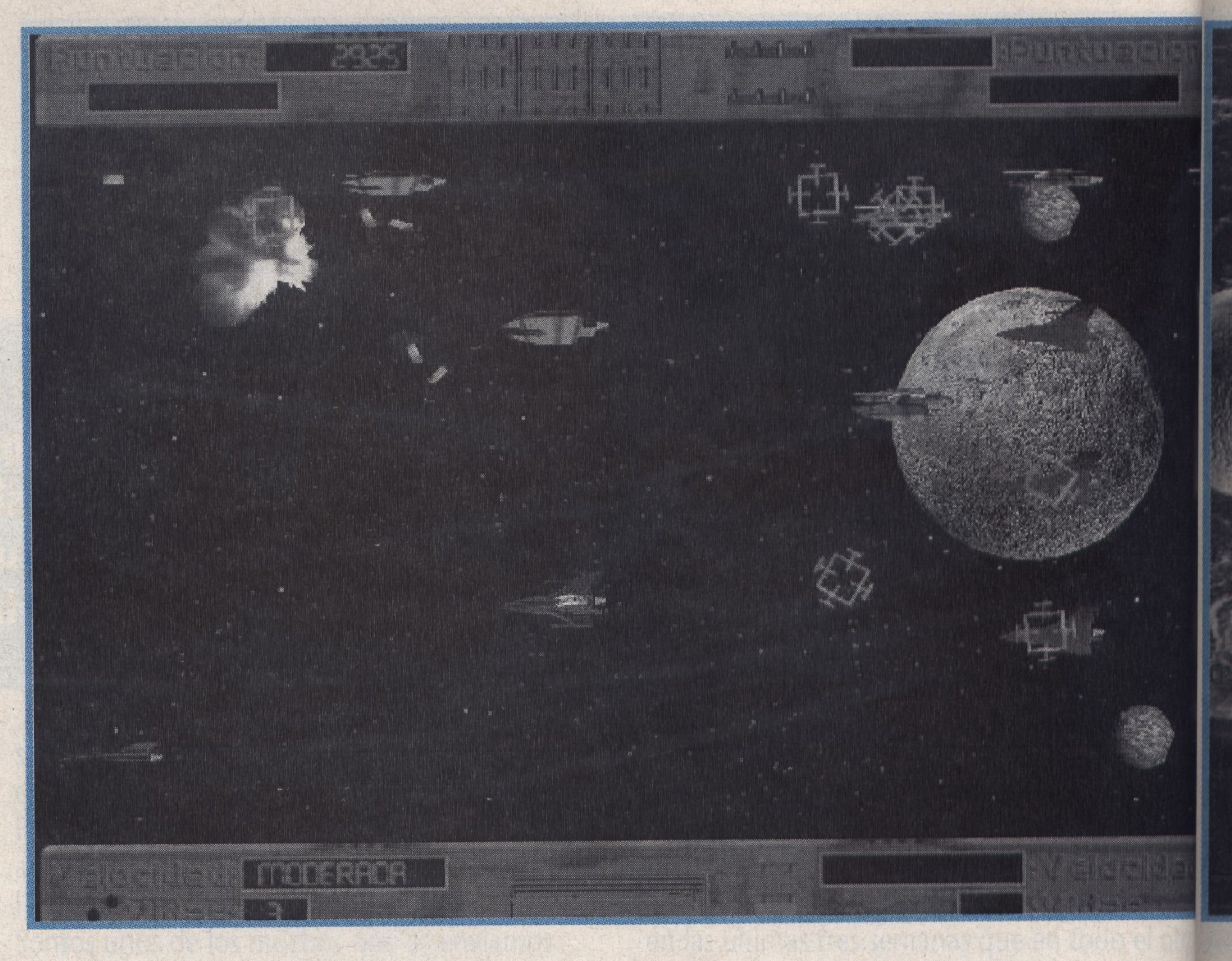
CANCION;           //ID del MOD que esta
sonando

T_ABAJO_P;         //Para cambiar de opciones
en los menus
T_ARRIBA_P;        //Para cambiar de opciones
en los menus

// Para cuando se cambia de KEYBOARD a PAD
STRUCT TEXTOS[15]
    STRING TEXTO[20];
END

// Estructura para el ranking
STRUCT NOMBRES[10];
    STRING NOMBRE[19];
```

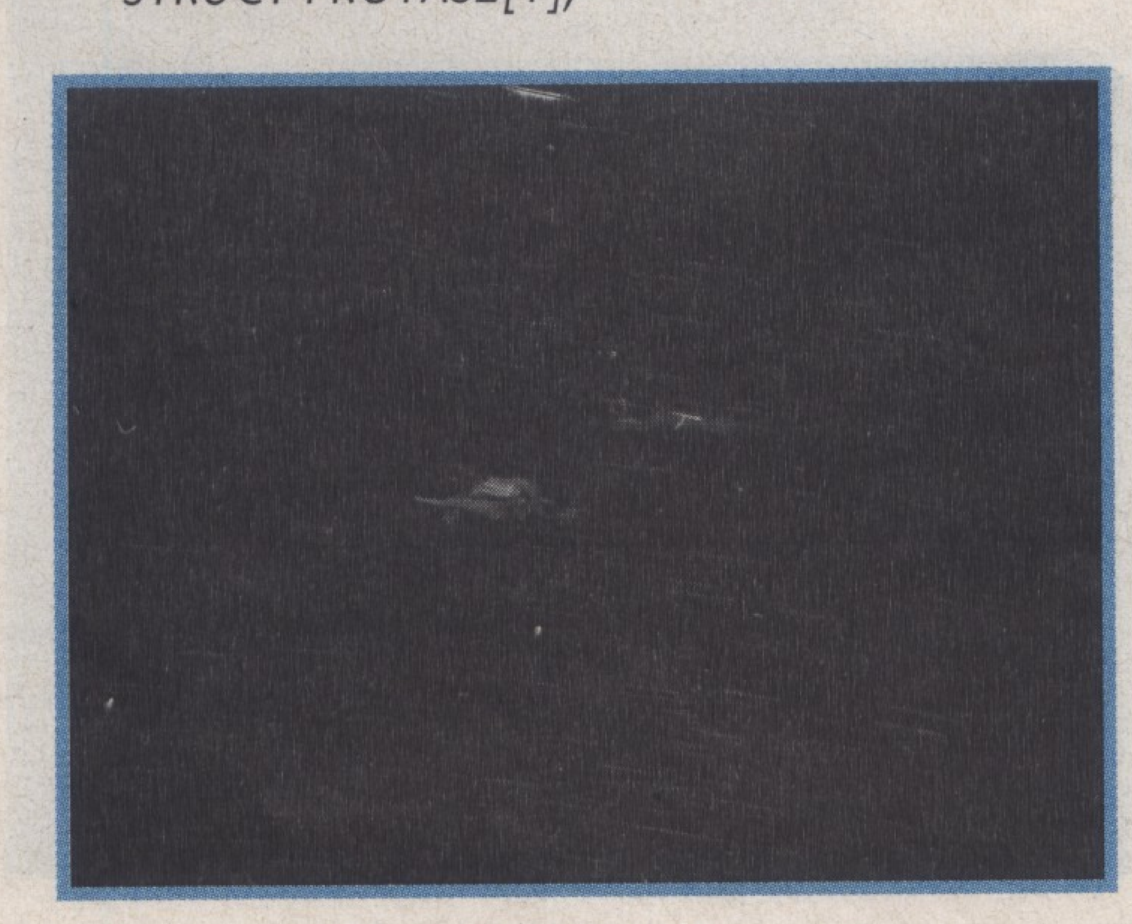





```
        PUNTOS;
    END

// Estructura para los controles de los protas(los
datos se guardan en F_CONTROLES)
    STRUCT PROTAS[1];
        X_INI;
        Y_INI;
        ARRIBA;
        ABAJO;
        DERECHA;
        IZQUIERDA;
        DISPARO;
        RECARGA;
        VELOCIDAD;
        CONTROLES;
        DISPARO_JOY;
        VELOCIDAD_JOY;
        RECARGA_JOY;
    END

// Estructura para datos temporales de los protas
    STRUCT PROTAS2[1];
```

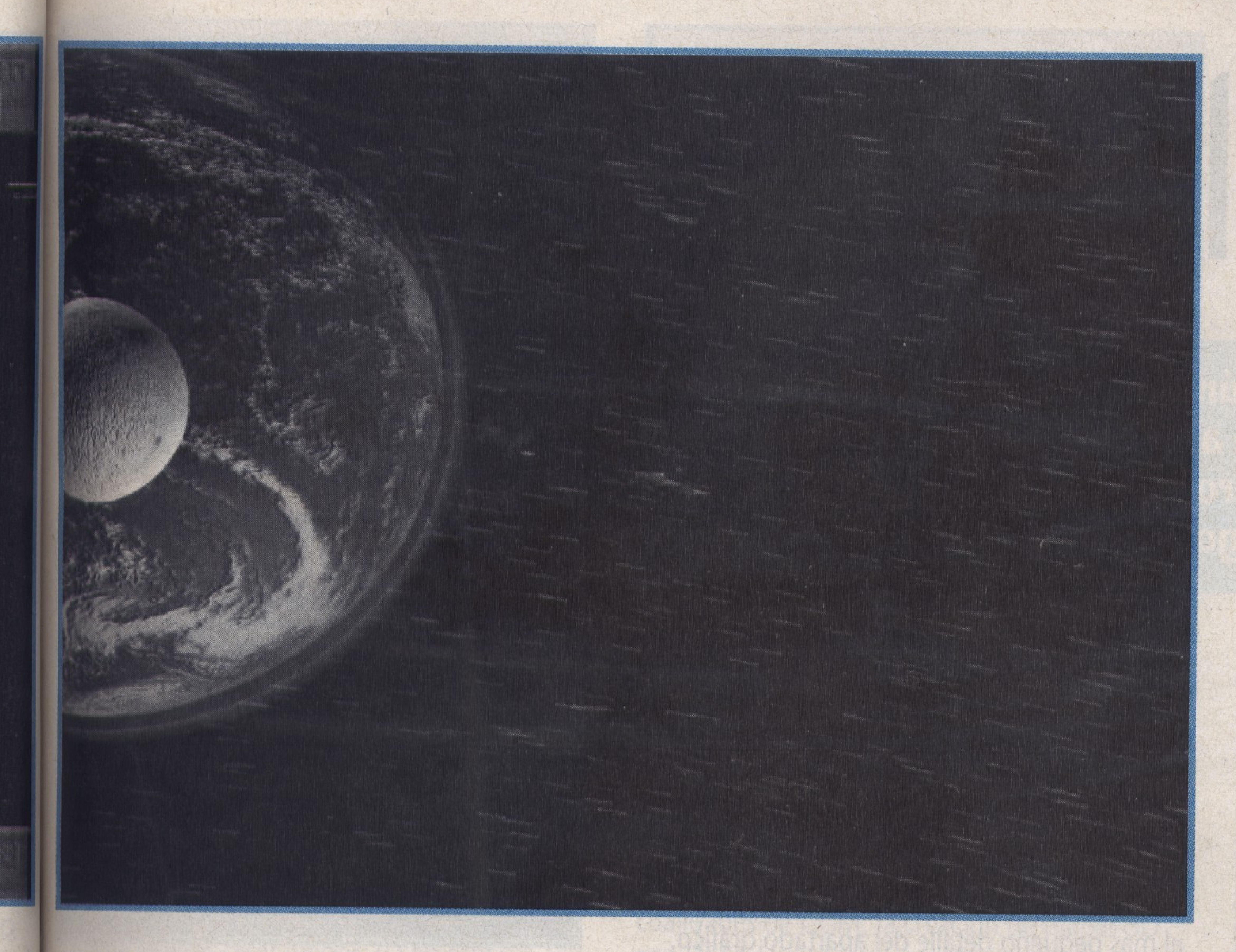


```
        ID_PROTA;
        PUNTOS;
        PUNTOSVIDA;
        TIPO_VEL;
        VIDAS;
        CONTINUES;
        RECARGA;
        TEXTO_VEL;
        MUERTO;
        NAVE;
        TIPODISPARO;
        MISILES;
    END

// Estructura para los tipos de
disparo
    STRUCT DISPAROS[4];
        GRAFICO;
        GRAFICO2;
        DA¥O;
    END= 101,99,2,
        96,95,1,
        92,91,1,
        93,94,3,
        90,89,1;

// Estructura para los misiles
    STRUCT MISILES[1];
        IDOBJETIVO[7];
    END

// Estructura para las trayectorias de los
enemigos
    STRUCT TRAYECTORIA[10];
        N_SECCIONES;
        X_INI;
```



```
Y_INI;
STRUCT TRAYEC[2];
   IMAGENES_POR_SECCION;
   VELOCIDAD_X;
   VELOCIDAD_Y;
 END
END=
     3,660,80,
     54,-10,0,
     32,0,10,
     1000,10,0,

     3,660,380,
     46,-10,0,
     28,0,-10,
     1000,10,0,

     3,660,240,
     45,-10,0,
     5,0,-6,
     1000,10,0,

     2,660,240,
     45,-10,0,
     100,10,0,
     0,0,0,

     3,660,80,
     20,-10,0,
     28,0,10,
     1000,10,0,

     3,660,380,
     12,-10,0,
     28,0,-10,
     1000,10,0,

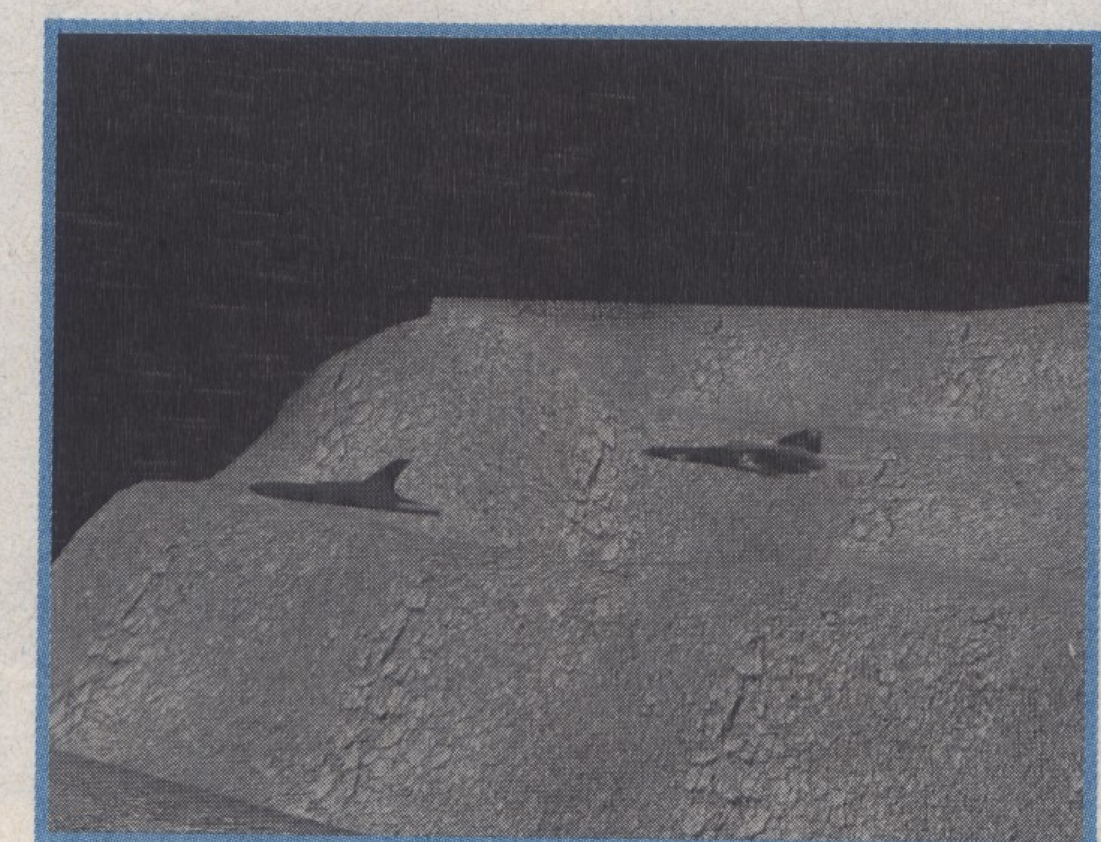
     3,-20,80,
     20,10,0,
     28,0,10,
     1000,-10,0,

     3,-20,380,
     12,10,0,
     28,0,-10,
     1000,-10,0,

     3,660,80,
     20,-10,0,
     50,-5,6,
     1000,-10,0,

     3,660,380,
     20,-10,0,
     50,-5,-6,
     1000,-10,0,

     3,660,200,
     20,-4,-2,
     150,0,1,
```



```
     100,4,-2;

  //POSICIONES DE LAS DIFERENTES PARTES
DEL JEFE
   XCA¥;YCA¥;
   XLAS1;YLAS1;
   XLAS2;YLAS2;
   XDISP1;YDISP1;
   XDISP2;YDISP2;
   XDISP3;YDISP3;

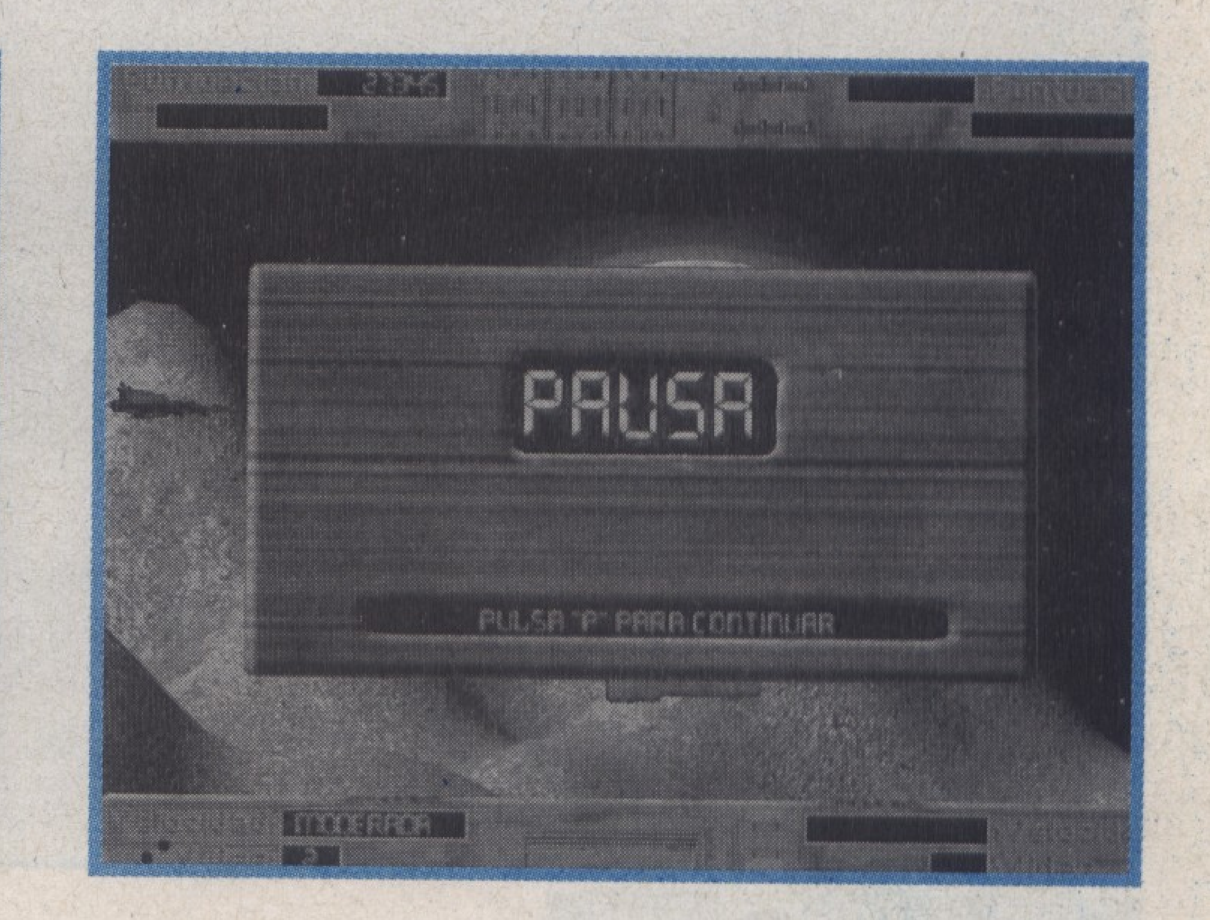
LOCAL

   ENERGIA;      //Energia de los enemigos
   ID_AUX;       //ID auxiliar usado en varios
procesos
   ESTADO;       //Estados de diferentes procesos
   RECARGANDO;
   X1[3];Y1[3];  //Posiciones de los reactores
respecto a su padre

BEGIN
  set_fps(20,1);
  set_mode(m640x480);
  KATANA();
END

//----------------------------------------------------------

PROCESS KATANA()
BEGIN
  set_fps(15,0);
  start_fli(FLIKATANA,0,0);
  fade(100,100,100,3);
  REPEAT
    FRAME;
  UNTIL (!fading);
  REPEAT
    X++;
    FRAME;
  UNTIL (frame_fli()==0 | key(_esc))
  FOR (z=0;z<15;z++)
    IF (x<199) BREAK;END
    FRAME;
  END
  fade(0,0,0,1);
```





Autores: Carlos Gª Mérid
y Joan Collad

Dark Castle

El segundo premio de este número de nuestra revista tiene un carácter especial. Más que a un solo juego, el premio va a la labor de un prolífico Divero que nos hizo llegar toda su colección de juegos. Enhorabuena.

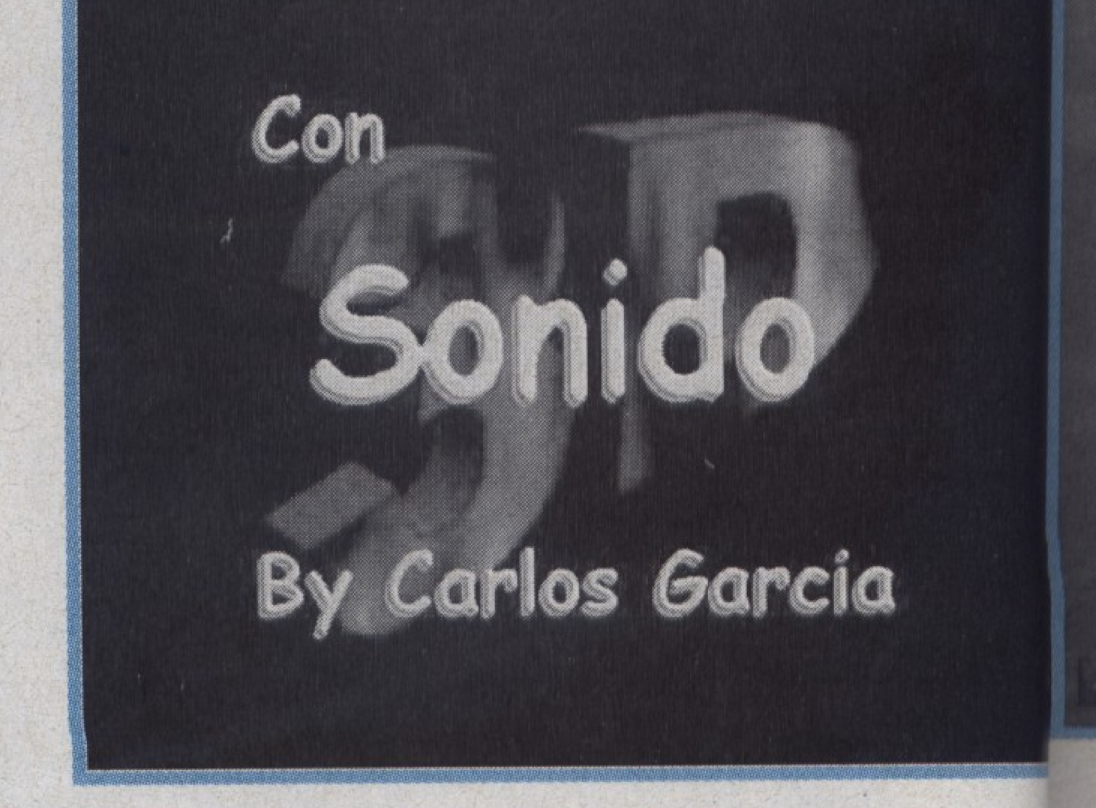


Lo cierto es que nos sorprendió encontrar un Cd con tantísimos juegos y pensamos, a poco que alguno sea bueno se lleva algún premio, y así ha sido. El caso es que se lo hemos dado a *Dark Castle* pero se lo podíamos haber dado perfectamente a casi cualquier otro. Ninguno de ellos tiene desperdicio y todos son muy interesantes y entretenidos. Por otro lado, Carlos García también ha conseguido que sus juegos DIV tengan sonido 3D, algo que le ha dado muchas papeletas de cara al premio. Como siempre, os dejamos con las palabras de los creadores (que son los que mejor pueden hablar de su juego) para que nos hablen de *Dark Castle* y el resto de sus creaciones:

¡¡Ya esta aquí!! Esta es la primera creación de JCG software, y no está nada mal ¿verdad?. Como viene siendo habitual en nuestros temas, se trata de un juego 3D en primera persona. La calidad técnica del juego es innegable, desde el espectacular diseño de los escenarios, así como el más pequeño detalle del apartado gráfico. En cuanto a programación, los enemigos no tienen lo que se llama una IA, pero aun así lograrán sorprenderte, e incluso asustarte en algunas ocasiones. La música es la guinda que adorna el pastel, consiguiendo una más rápida introducción en la historia.

En este juego hemos querido salir un poco de la dinámica en la que nos habíamos metido, (matar a diestro y siniestro a todo aquello que se mueva) y profundizar un poco más en la diversidad que se puede conseguir a partir de un juego 3D.

De entrada la temática cambia bastante (no hay que matar sino evitar ser matado). El juego no tendría ningún gancho, si no fuese por el agravante de que todo esta oscuro y sólo



SONIDO 3D

¡¡¡¡ Por fin lo he conseguido !!!! He tardado, por diversas cuestiones, pero cierto es que lo he conseguido.Ha sido dificil, muchas comidas de olla, muchas pruebas en falso, etc... pero al final, el resultado compensa.
No es un SONIDO 3D perfecto, porque no distingue entre delante o detr pero por lo menos si entre los lados. (que ya es algo mas, aparte de la profundidad).

Dado que ha pasado tiempo, he hecho algunos cambios. Ahora, el SONI 3D pertenece a JCG SOFTWARE. El logo es distinto, y pido a todo aqu que utilice estas ecuaciones, que lo use. El programa de demostracion, mismo que el anterior ya que el sonido cambia constantemente, aunque podria aplicar en el primero tambien.



(c) 2000 JCG SOFTWARE

dispones de tu candelabro para ver (efecto conseguido con FOG) y los esqueletos, no so inteligentes (cuando chocan, eligen un núme del 1 al 8 y lo multiplican por 45 grados, es u manera de que vaguen sin sentido pero con poco de control) pero al aparecer de entre la oscuridad sin avisar, o al encontrártelo avanzando muy cerca de la pared, nunca sab lo que va a hacer, y eso es lo que mantiene a juego algo interesante. Creo que incluso pod decirse que es algo adictivo, con algo especia que hace que vuelvas a jugar.

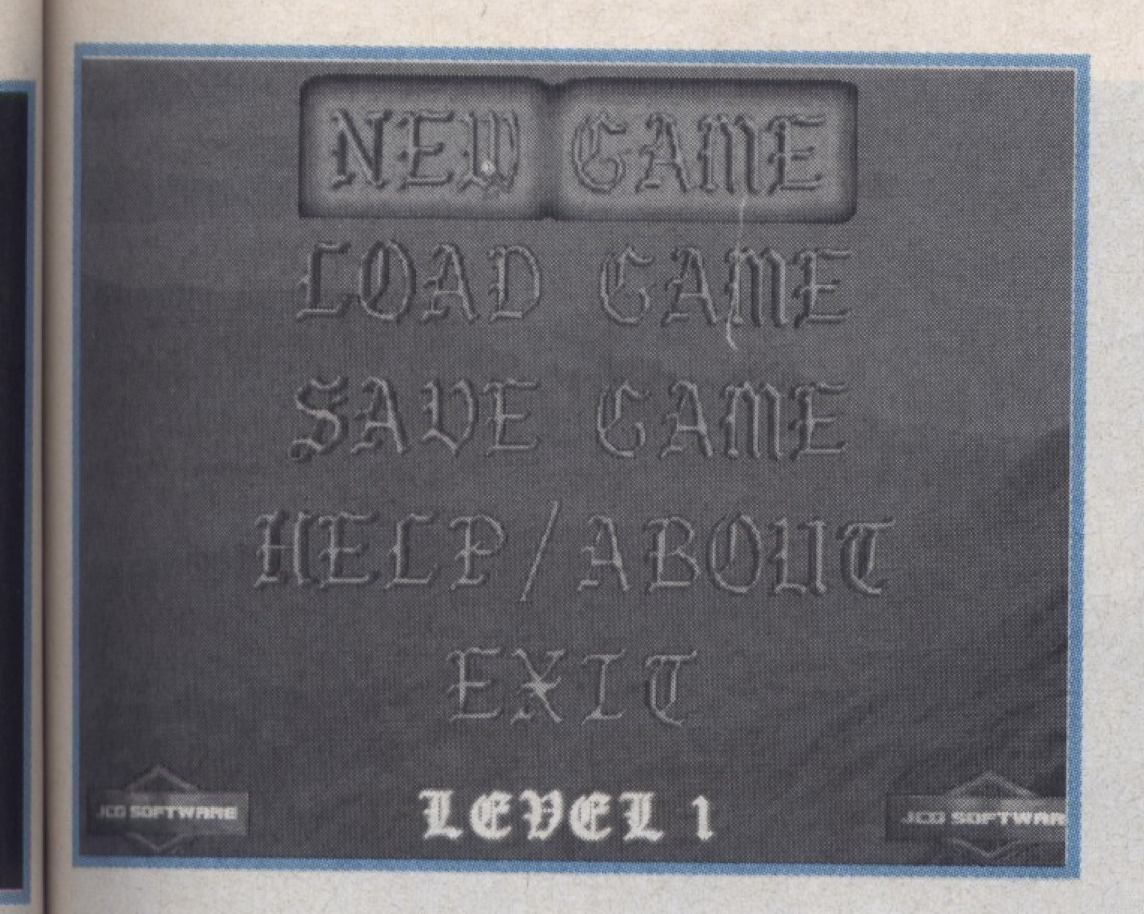
Hemos encontrado algunos fallos en nuestro juego difícilmente subsanables, el más significativo es que a veces, sobretodo en el primer nivel, vas andando tan tranquilament de repente te dice que el objeto ha salido de mapa, y el programa, o se cuelga, o vuelve a menú. También encontraras de vez en cuan a algún esqueleto atontado, en una esquina hacer ningún tipo de movimiento (¿por qué No lo sabemos).

El juego en si, consiste en ir a los sitios indica en el minimapa, y recoger los objetos sagrad cuando estén todos debes dirigirte al lugar de salida y tocar la correspondiente pared. Así d fácil. El ultimo nivel es idéntico solo que al co los objetos, aparece un pico del suelo y eso e que tienes que tocar. El movimiento se cons con los cursores y junto con la tecla ALT se consigue un efecto STRAFE.

ÉRASE UNA VEZ

La historia de este grupo de programación que nos ha hecho llegar sus juegos es de l





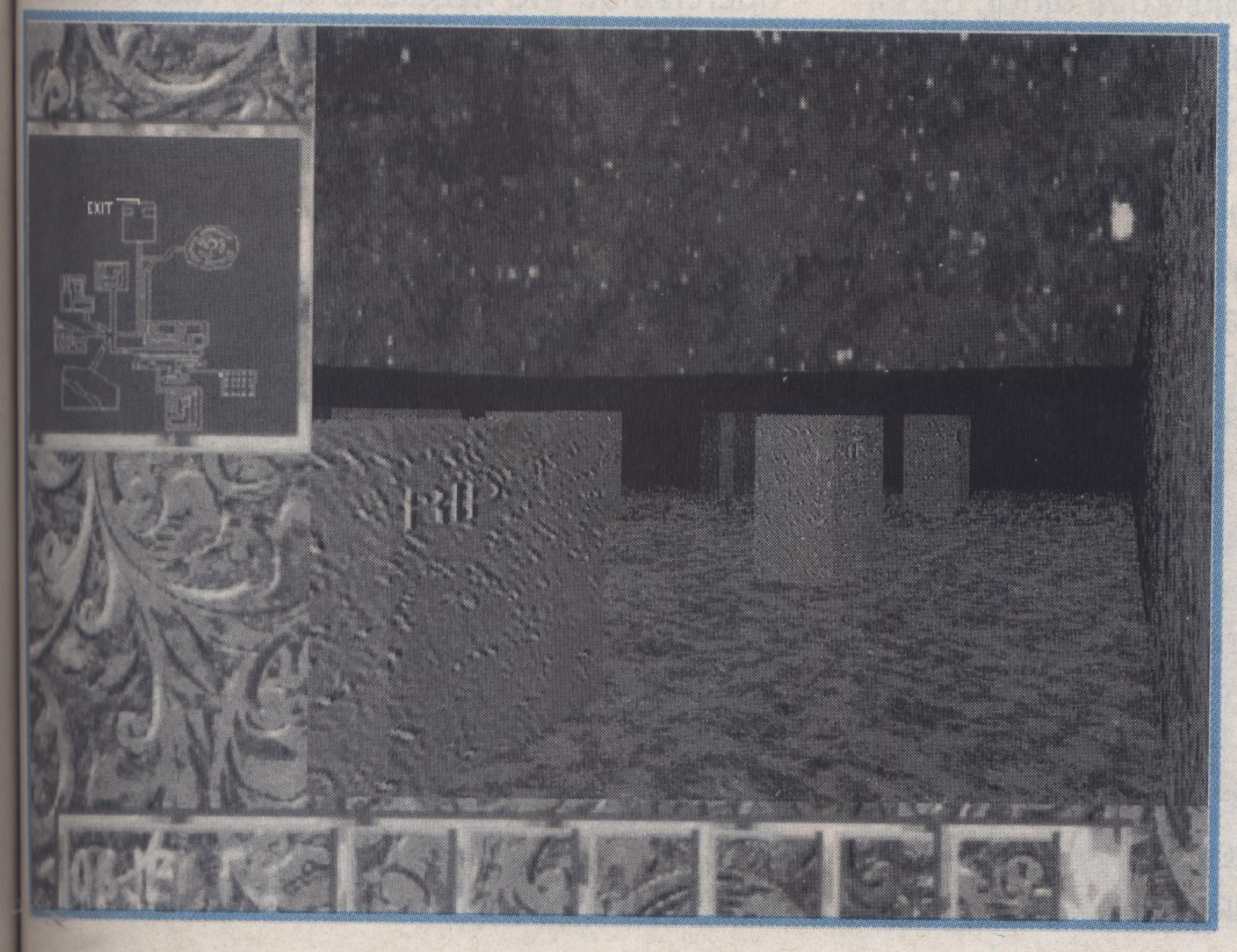
MIENTRAS VISITAS LAS TUMBAS DE TUS ANTEPASADOS, ALGO EXTRANO OCURRE A TU ALREDEDOR... DE REPENTE OSCURECE.

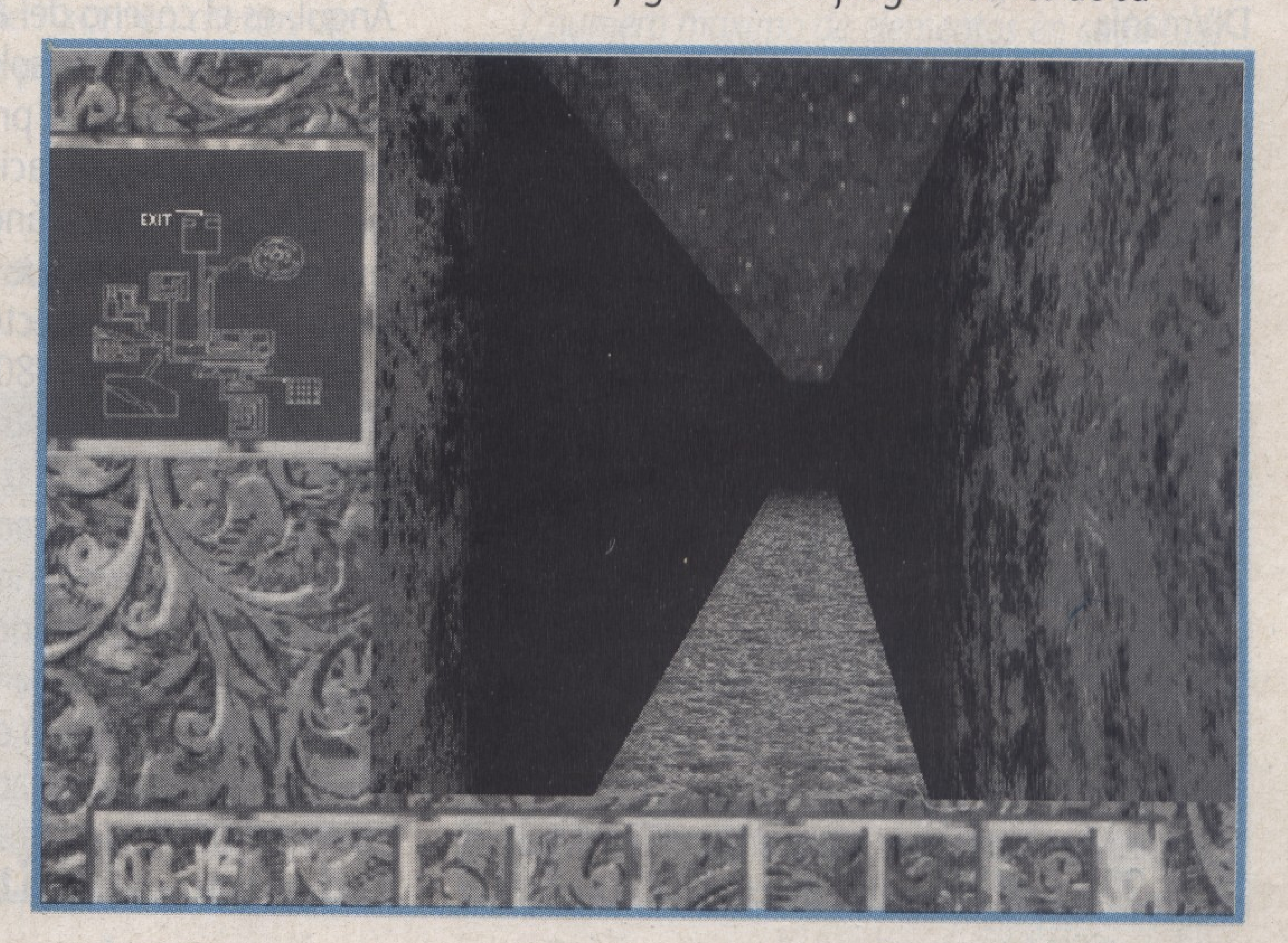
A PENAS CONSIGUES VER CON AYUDA DE TU CANDELABRO. POR SUERTE TIENES UNA BUENA RESERVA DE VELAS.

NO ACIERTAS A EXPLICAR QUE ESTA PASANDO. MIENTRAS PIENSAS OYES UN RUIDO EXTRANO Y TERRORIFICO. ALGO SE ACERCA A TI ...

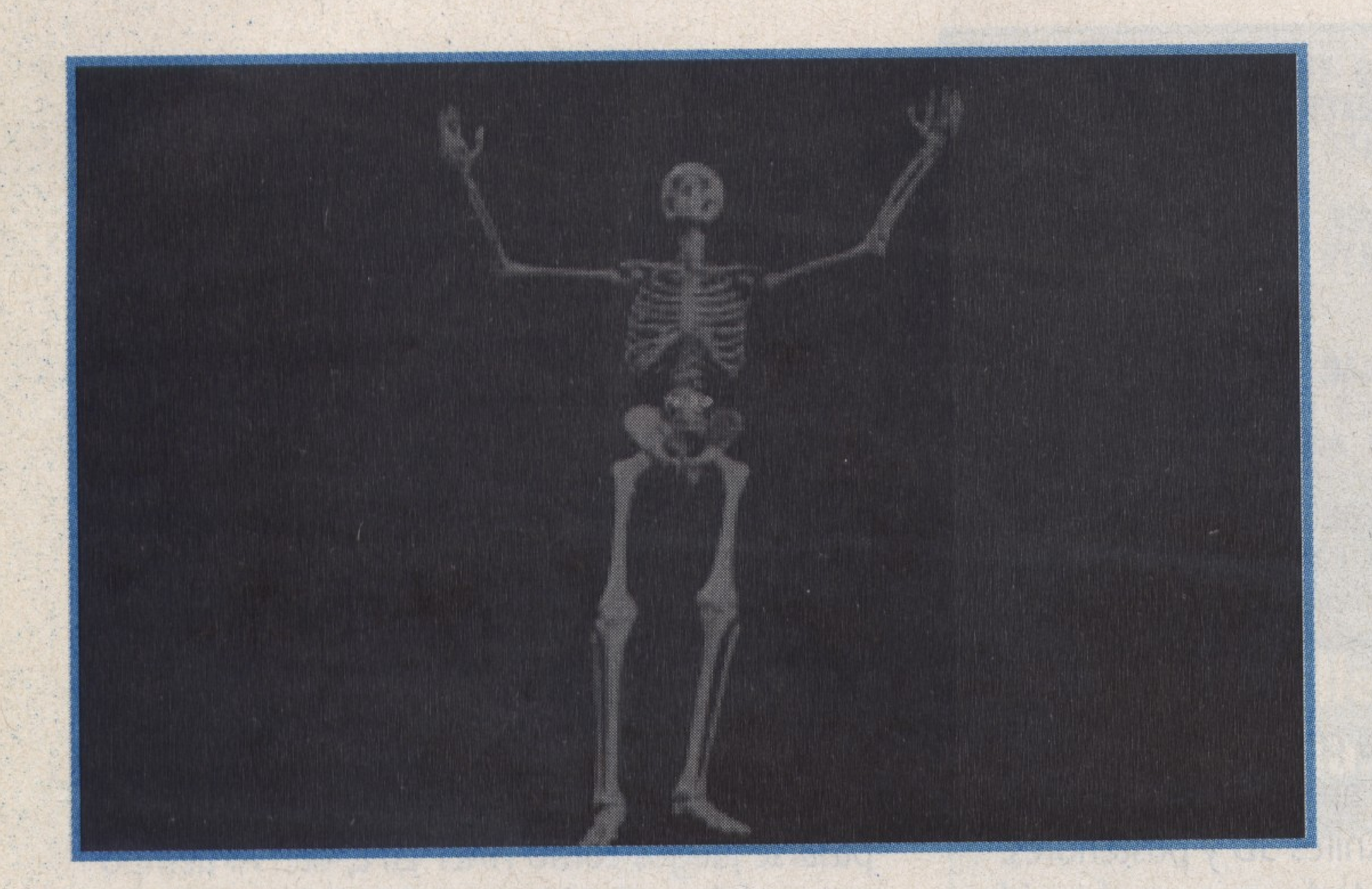
POR DESGRACIA NO TIENES NADA PARA DEFENDERTE. ASI QUE OPTAS POR APARTARTE. REZANDO PARA QUE NO TE VEA.

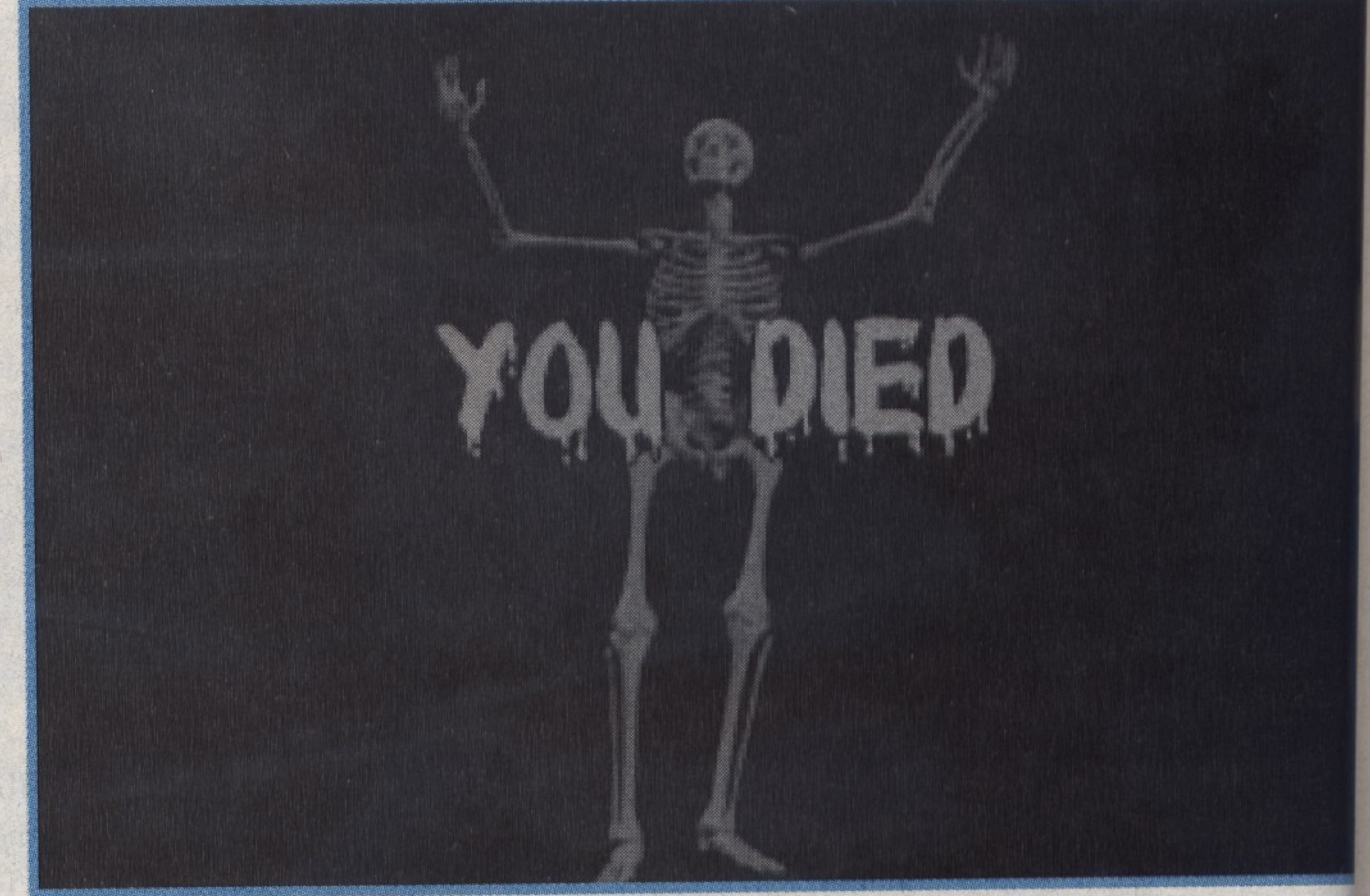
más curiosa y no nos hemos resistido a contarosla. Allá va.
CGM se compra DIV2 (conocía DIV 1 pero no lo tuvo) y empieza a programar sus propios juegos. Empieza CGM Software. El primer titulo de CGM Software, The revenge of the Nabo, es una conversión de un juego de muestra que viene con DIV2, pero en versión X. No se recuerda cuando fue hecho, pero aproximadamente en octubre del '99. Cuando CGM empezó a profundizar en el campo de la programación con DIV descubrió que era bastante sencillo crear juegos 3D en este lenguaje, así que comenzó su andadura por las tres dimensiones de DIV2.
El primer juego 3D tampoco lo hizo del todo CGM, era (una vez más) una adaptación de un juego, esta vez conseguido a través de DIVmanía que servía como ejemplo de un curso de 3D en la revista electrónica DIVnet. El juego es llamado Knifes 3D, y no tiene más argumento que empezar a matar esqueletos a diestro y siniestro. El juego fue concebido aproximadamente dos semanas después del lanzamiento de DIVmanía nº4. Al mismo tiempo, nació la idea de crear el sonido 3D, dado que en el juego Knifes 3D beta 1, Al matar un esqueleto muy lejano, sonaba como si lo hubieras matado al lado tuyo. Después de muchas comidas de olla, se consiguió dar con la formula perfecta para conseguir un efecto de profundidad de sonido en DIV2. Esta técnica fue implementada en Knifes 3D y posteriores. Pero faltaba la clave de todo, que era el sonido envolvente. Se sabía como convertir un ángulo, al valor que necesitaba DIV para conseguir ese efecto, pero faltaba el propio ángulo. Las investigaciones se pospusieron por falta de tiempo, y ansia de creación de juegos. Al tiempo nació JGun, el juego insignia de CGM Software. En el se exhibían las mejores cualidades de CGM, la creación de mapas 3D con DIV2. El juego en si era bastante simple, una cita en un parque a las tantas para liarse a tiros.
El juego, poseía la tecnología del Sonido 3D en su fase de profundidad. Por aquella época, era todo un alarde de tecnología de construcción y desarrollo en DIV2 por parte de CGM.
El juego gustó mucho (a CGM y sus amigos) y por eso se creó su secuela. Casi sin descanso, CGM se puso a trabajar en dos juegos simultáneamente, a la espera de que las negociaciones con Joan Collado llegasen a buen puerto, y se comenzase con la distribución de los juegos. Los juegos eran, el esperadísimo Jgun2, secuela del aclamado JGun, y el alocado Drivin' Wild, una adaptación DIV del clásico del gore divertido, Carmaggedon.
Por estas fechas, se tenia casi claro que se enviaría a DIVmanía un CD con los tres primeros juegos de CGM, pero debido a problemas ajenos a nuestra voluntad, el proyecto tuvo que ser anulado. Los archivos de texto estaban ya grabados en el CD junto con otras cosas, pero un problema técnico en el sistema de transporte de un ordenador a otro, imposibilitó acabar el primer intento, que tuvo que posponerse hasta el año 2000. En ese tiempo, CGM se dedico a sus dos proyectos comenzados JGUN2 y DRIVIN, pero al tenerlos prácticamente acabados, se cansó de ellos, y los abandonó momentáneamente. Llegó una época de sequía programadora, a favor de pasarse juegos como HALF LIFE, etc. Al tiempo se reanudaron las ideas en CGM. Se creó Tae Kwon Do (vamos, se adaptó de nuevo una muestra del DIV2) en honor a Miguel Angel el primo de CGM (Si queréis saber más cosas del él, poned al juego).
Después, CGM retomó de nuevo los dos proyectos olvidados, consiguiendo acabarlos en poco tiempo. Ya se disponía de material de sobras para enviar a DIVmanía así que se procedió a la grabación del CD. Cuando parecía todo claro. El retraso era tal, que a la revista, le faltaba muy poco tiempo para aparecer en el mercado, por lo que decidimos esperar al siguiente número, y llenar el CD un poco más con nuevas ideas. En ese plazo de prorroga, nació oficialmente JCG Software, y terminaron las andaduras en solitario de CGM. Un nuevo proyecto en 3D nacía, quizá el más ambicioso, y quizá el mejor candidato a ganar el concurso de DIVmanía, DARK CASTLE. La idea desde un principio era buena, esto, mezclado con el buen manejo de CGM con el generador de mapas 3D, dotó al juego de un impacto visual, nunca antes visto en DIV2. En lo referente a la programación, el juego no es una maravilla, (los enemigos no tienen inteligencia) pero se logra meter al jugador en el juego a través de su











ambientación oscura, su música muy acorde con la historia, etc. y se consigue una jugabilidad bastante aceptable. (a pesar de que el juego es relativamente corto). El juego es catalogado por CGM como la obra maestra de las 3D en DIV. Al tiempo que se programaba este juego, se pensó en el mismo tipo de juego, pero de un tipo más arcade, con más acción instantánea, así nació FastDie. El juego no tenia más trama que la de cercar en un espacio pequeño a decenas de esqueletos y a ti, con la única misión de sobrevivir el mayor tiempo posible. La verdad es que pica bastante intentar superarse.

A partir de este momento, CGM entro en una época de sequía de ideas en 3D, así que optó por versionar diferentes juegos 2D que le habían parecido interesantes. El primero de ellos fué FROGS, una adaptación del Bzzz de Astraware. (incluso usa su fondo de pantalla). El juego fue rápido de programar, la idea era parecida pero con variaciones, y el resultado es bastante adicctivo.

El segundo juego versionado, es DEADDUCKS, que es una versión de un juego de caza que conseguí en una revista, aunque aprovechando gráficos del juego de Tiro al Plato que venía en DIVmanía.

La grabación final del CD parecía inminente, así que ese parecía el ultimo juego de JCG, pero de repente CGM se sacó de la manga una maquina tragaperras. Este juego es el único hecho por CGM que tiene una historia interesante, que paso a relatar: "El juego nació en SEAT, (si, la fabrica de coches, como los ESTOPA), concretamente en el taller 9 de la fabrica de Martorell. Solo estuve un día allí y mi trabajo consistía en arreglar un error de chapa en el modelo AROSA, solo tenia que golpear en un sitio con un martillo en cada coche (pasa un AROSA cada 8 minutos aproximadamente), así que le pedí al encargado que me diese unas hojas para hacer algo, y en un rato escribí lo que sería un juego de tragaperras en DIV. Lo cierto es que el proyecto me pareció bueno, pero al pensar en que tendría que hacer todos los gráficos, etc. lo aparté por momentos... Hasta ahora, que como sobraba tiempo hasta enviar el CD, me decidí a hacerlo. Hasta aquí la historia de CGM.

SONIDO 3D

Sin duda una de las características más destacadas de los juegos de este grupo es el sonido 3D por lo que no podemos pasarlo por alto. Como antes, os dejamos con su creador:

Por fin llegó el auténtico sonido 3D envolvente, después de apartar el proyecto por falta de tiempo y de claridad de ideas, decidí retomarlo, con energías renovadas, y en un rato conseguí descifrar el enigma (a veces, olvidarte de un problema te ayuda a encontrar la solución).

La verdad es que no tiene desperdicio, se puede distinguir perfectamente cuando estás escuchando por un oído o por el otro. Ahora paso a detallar las características de esta nueva ecuación:

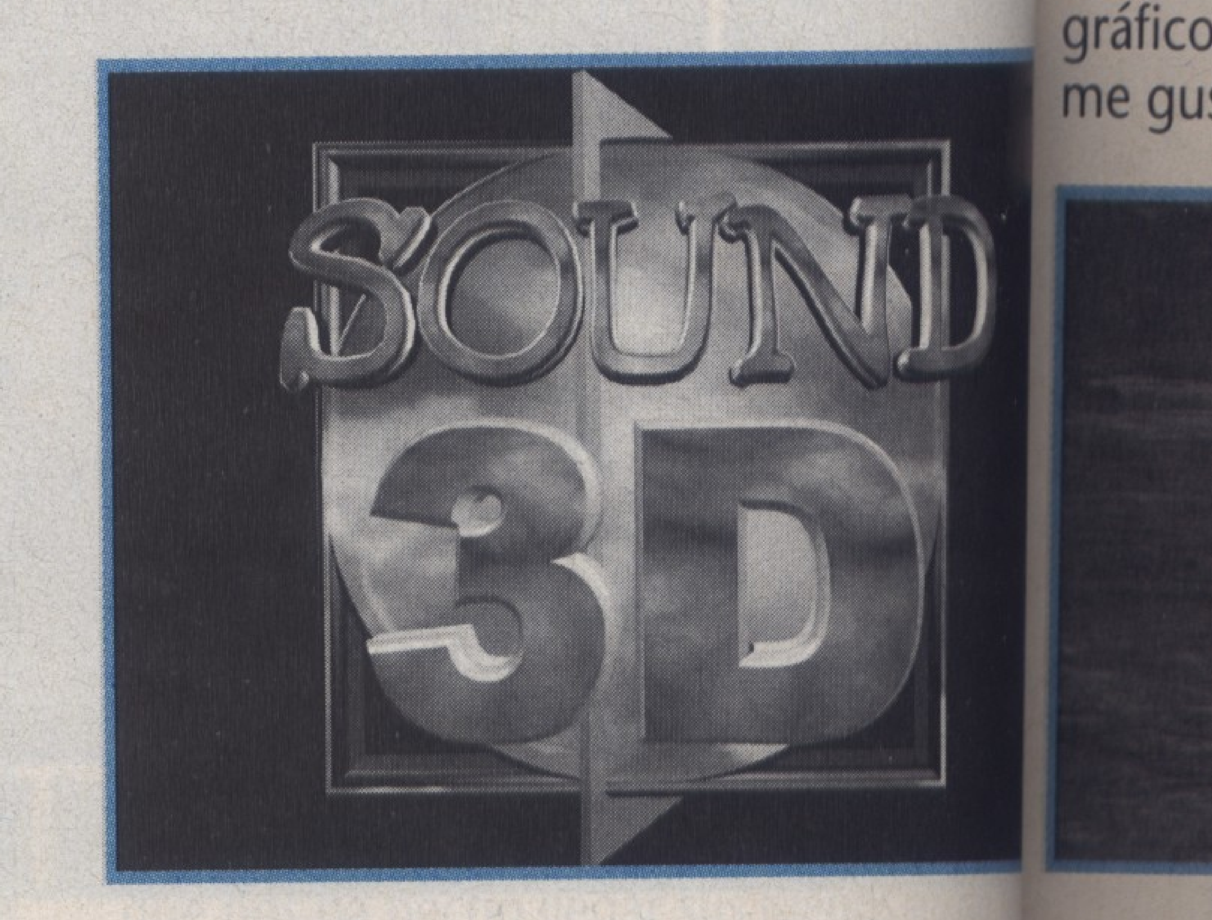
```
angul=(cos(((get_angle(pers)+180000)-
90000)-pers.angle))*256;
bal=256+((angul/1000));
change_channel(canal,vol,bal);
```

Angul, es el coseno del ángulo que tanto buscaba. Encontré la solución después de tanto tiempo, enfocando el problema desde otra perspectiva. La explicación sería que get_angle(pers) es el ángulo que forma la línea que une el proceso que oye al proceso que emite el sonido, respecto a una horizontal imaginaria (le sumo 180000, porque el valor que DIV retorna, es entre 180000 y -180000, y yo necesito uno entre 0 y 360000) . A ese ángulo, le quito 90 grados, porque el valor que busco con el coseno, necesita empezar en 0, y el coseno esta desfasado 90 grados (creo que esa es la explicación, la verdad es que ese valor lo encontré tanteando en las primeras pruebas, porque el valor de balance que obtenía, estaba desfasado).Y finalmente, le resto el ángulo que hace el que oye con la horizontal.

Así obtengo el ángulo que yo quería, (es un poco lioso a la hora de explicar, pero en el momento de razonarlo, lo tenía bastante claro, al parecer funciona, así que....), al cual le aplico el coseno para que me de un valor entre 1 y -1. Este valor, lo multiplico por 256, para obtener valores entre 256 y -256 (luego lo divido por mil, por que DIV trabaja con milésimas). Ese valor entre 256 y -256 se lo sumo a 256 (que el valor medio del balance en DIV) para obtener balances entre 0 y 512. Ya solo queda aplicar cambio con la función Change_channel. (donde vol se pone el valor de la profundidad de sonido del anterior sonido 3D y así se consigue el efecto completo).

Espero que hayáis flipado tanto como yo a la hora de hacerlo, y que os deis cuenta de que un gran avance para los programadores de D por ello os pido que si usáis la ecuaciones, utilicéis también el nuevo logo en vuestros juegos. Gracias anticipadas.

PD: Estas ecuaciones son freeware (faltaría ma así que todo el mundo puede usarlas, pero qu quede claro, que a pesar de no haberlas patentado ni nada parecido, el creador soy Y (Carlos García Mérida). Si tenéis alguna duda queréis algo (no necesariamente relacionado con el sonido 3D) escribidme a piyulaco@teleline.es o a Carlos Garcia Merida C\ Virgen de Montserrat nº 23, La Torre de Claramunt, Barcelona.



Autor: Juan Antonio Malo Poyatos

Laberyn

El tercer juego ganador es *Laberyn*, un shoot´em up que recuerda a *Wolfstein 3D*, tanto por sus gráficos como por su modo de juego: tendremos que recoger las llaves que se nos piden para acceder a otras fases y matar a todo aquel que intente impedir nuestro progreso. La jugabilidad es lo que más destaca en esta obra.

Este juego es muy parecido a aquellos primeros shooter que tanto nos hicieron disfrutar hace algunos años, ¿quién no recuerda *Wolfstein 3D, Doom, Hexen* o *Heretic?*, que tiempos, *¡o tempora o mores!* En fin, que muchas gracias por enviarnos tu juego y felicidades por el premio, sólo una cosa: quizás deberías buscar la ayuda de alguien que se dedique al diseño gráfico para que tus juegos tengan un aire más nuevo, los autodidactas en esto de la programación de juegos están en vías de extinción, de todas maneras nos hemos divertido un montón jugando con *Laberyn*, alguno incluso se ha emocionado al verse trasladado por un lapsus de tiempo a la infancia. Bueno, ya está bien de sentimentalismos, os pasamos a Juan Antonio.

EL AUTOR Y SU OBRA

Hola, que tal estáis. Me llamo Juan Antonio Malo Poyatos, y llevo en esto de la programación desde hace bastantes tiempo, desde que comencé con *BASIC*, luego *C/C++*, *Pascal*... pero aunque he hecho programas de distintos tipos, hasta que llego DIV no he tenido tanta facilidad para hacer juegos. *Laberyn* es un juego con visión 3D, como el *Doom*, de hecho algunas de las texturas son de este juego, pues así ahorraba tiempo en gráficos y me dedicaba más a lo que realmente me gusta, programar. Consta de 3 fases, cada una de las cuales se pasa consiguiendo un determinado número de llaves para abrir la puerta de acceso al siguiente nivel.

El disparo es simple, se basa en un calculo por distancia, distinta para cada arma: si esta el objetivo dentro del área, le das aunque no le veas. Lo s,, es un algoritmo un poco tosco, pero es que he simplificado al máximo, y va aceptablemente bien.

Algunas paredes pueden dar problemas de visibilidad: si aparecen, moverse en dirección contraria. Asimismo, como el juego lo he realizado en un 486 a 50Mhz y 8 MB, solo he podido probar el juego en el modo 320x200, en donde va bien. El modo de alta resolución 640x480 no lo he probado demasiado, aunque en teoría no debería dar problemas, pues solo cambia la resolución.

En mi cacharro va lento, pero en un ordenador de un amigo, un Pentium a 200 Mhz con 32 MB de RAM, con tarjeta gráfica de 1 MB es rápido y muy jugable.

Como es un juego muy corto, no se puede salvar la partida, por lo que hay que hacerse el juego del tirón, se puede hacer en menos de media hora. Espero que por lo menos paséis un buen rato.

CONTROLES

Barra espaciadora: usar objetos, abrir puerta, usar ascensor.



Control: disparar el arma que se tenga seleccionada.
Flecha arriba: Avanzar adelante.
Flecha abajo: Retroceder.
Flecha derecha: Girar a la derecha.
Flecha izquierda: Girar a la izquierda.
Alt + cursor: traslación sin giro.
Shift + cursor: movimiento rápido.
1: Seleccionar la pistola (si la has conseguido)
2: Seleccionar el rifle (si lo has conseguido)
0: Seleccionar sin arma, puñetazos.
Esc: escape rapido, equivale a Alt-X.
Alt-X: aborto normal.

CÓDIGO

```
COMPILER_OPTIONS _MAX_PROCESS=800;

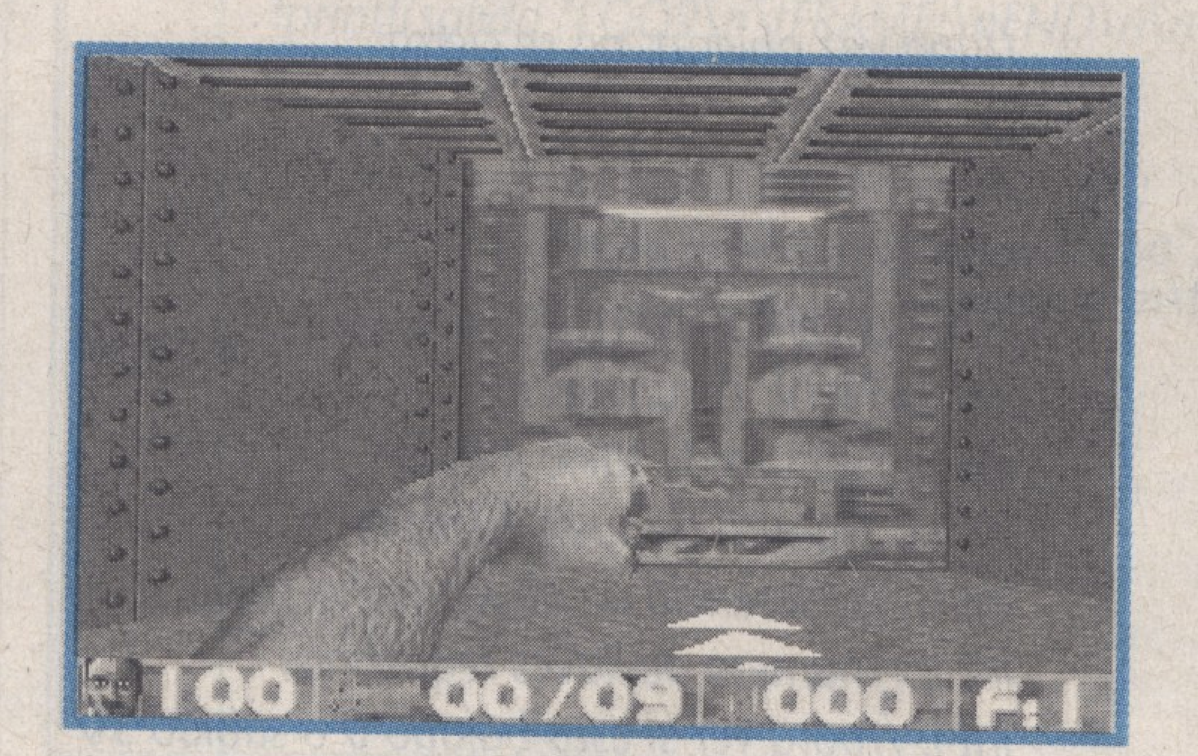
PROGRAM laberyn;

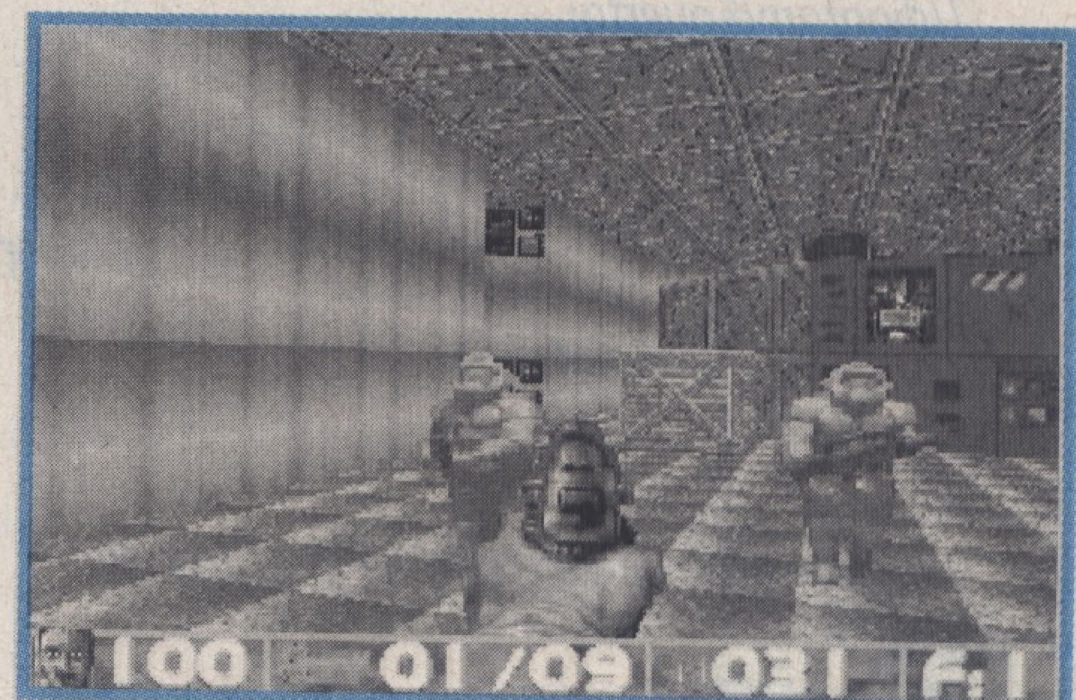
CONST
   ARCHIVO_WLD = "mios\laberyn1.WLD";
   ARCHIVO_FPG = "mios\laberyn1.FPG";
   MUSICA="mios\laberyn1.xm";
   SONPUERTA="mios\puerta2.pcm";
   SONLLAVE="mios\cogelav2.pcm";
   DIST=200; //distancia minima para abrir
puertas
   DISP_RIFLE="mios\dispar04.pcm";
   DISP_PISTOLA="mios\pistola9.pcm";

   //numero maximo de elementos en cada fase
   num_puertas=15; //numero de sectores puerta
   num_puertas_bis=15; //numero de sectores
puerta con bisagras
   num_ascensores=5; //numero de ascensores
   num_teletransportadores=5; //numero de
teletransportadores
```

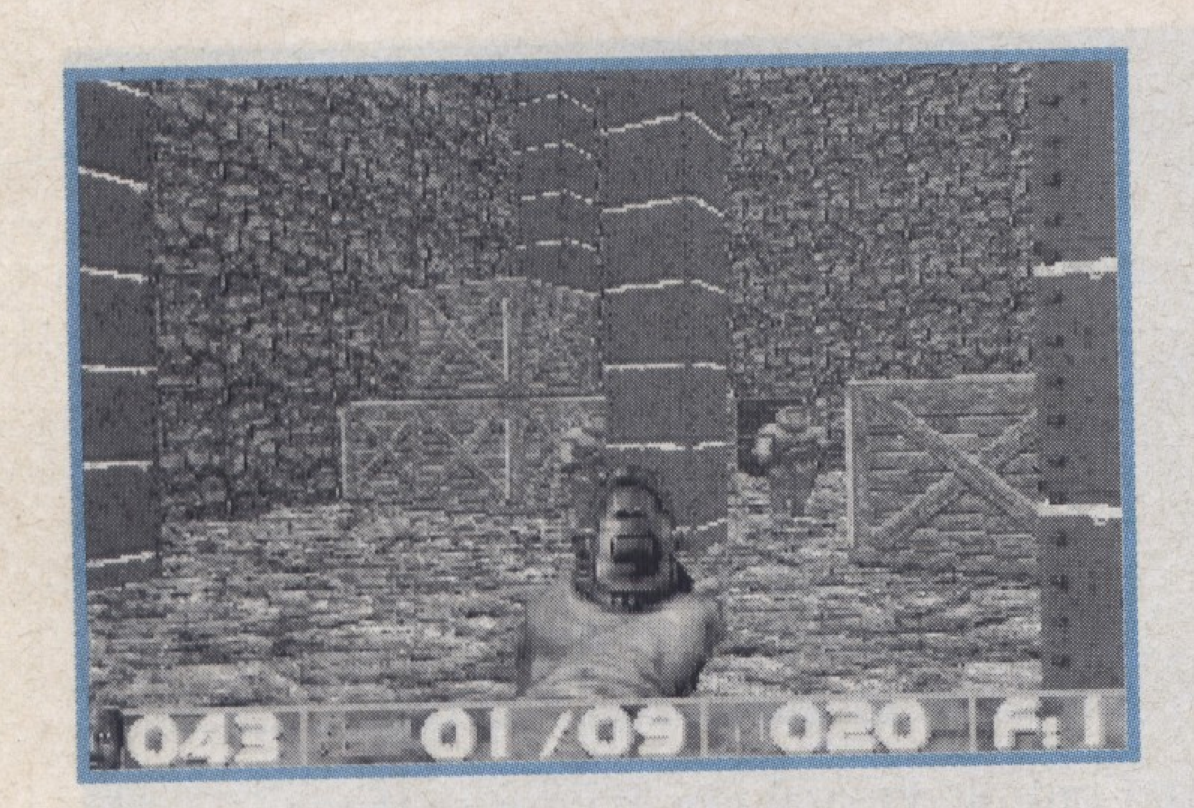
Tu última presa era un tipo de la peor calaña, psicópata y muy peligroso: EL MASTER, loco por el rol, responsable de multitud de asesinatos al azar que creía necesarios para completar una partida imaginaria en su cabeza pero que él había trasladado a la vida real.

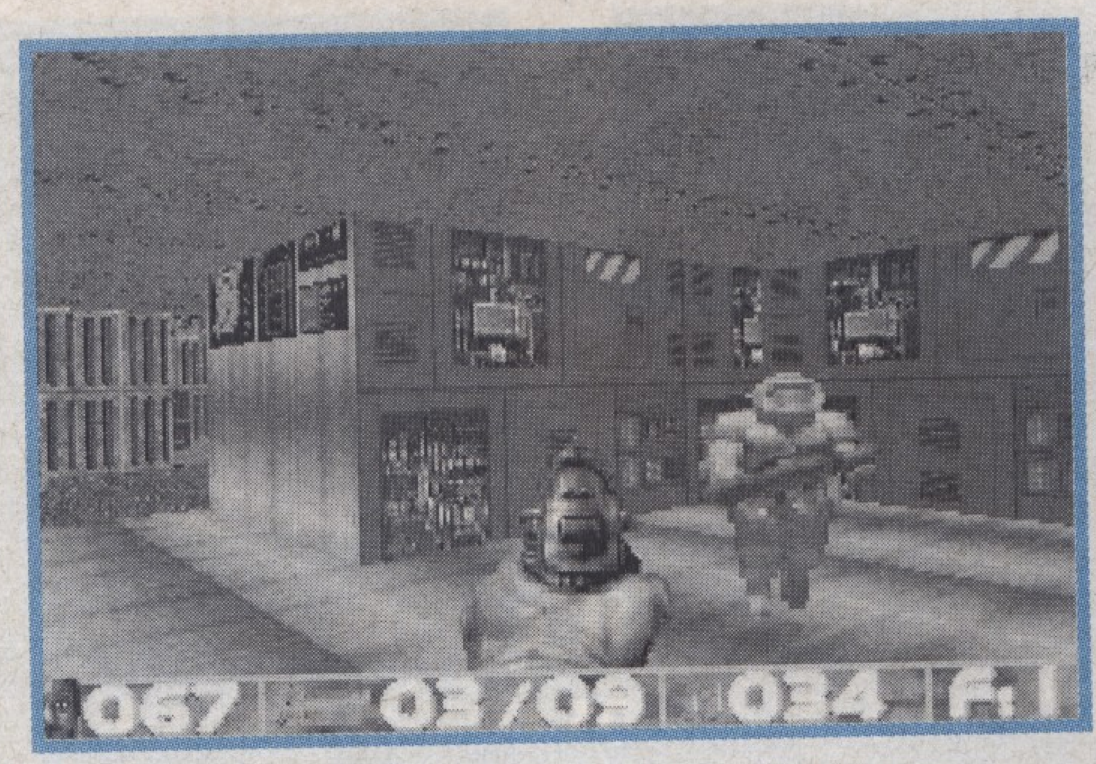
Furioso por haber eliminado a varios de sus 'colegas' de la 'partida', decide estrenar un nuevo 'módulo' pensando en su jugador favorito: TU. Y para que no dejaras de acudir a su

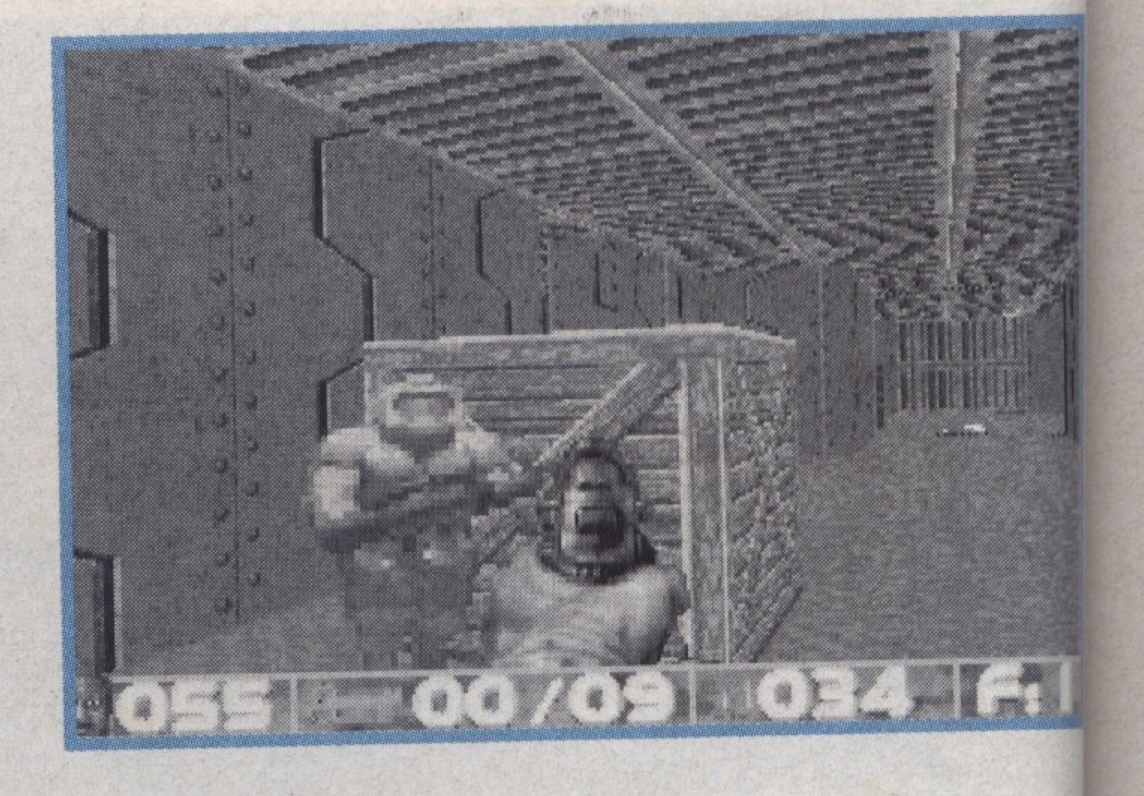




Juegos ganadores 3º








```
GLOBAL

    int final=0;
    //identifica al jugador
    playerid;

    int llaves=0;

    id_sonpuerta;
    id_sonllave;
    id_musica;
    id_sondisp_rifle;
    id_sondisp_pistola;

    int tecla;

    int i,j,k,q;
    int nn, nup; //para luminosidad de teletransp.
    int distancia,subirp,subira;

    //tabla de sectores puerta
    //inicializo .n=9999 para que por defecto
    //no sea 0 y no afecte al sector 0
    STRUCT puerta[num_puertas]
        int n=9999; //num sector
        int x;
        int y;
        int s; //altura de suelo de sector
        int t; //altura de techo de sector
        int l; //numero de llaves requeridas
        int i; //para ver si esta bajada 0 o subida 10
    END
    //NUEVO puertas con bisagras
    STRUCT puerta_bis[num_puertas_bis]
        int n=9999; //num sector
        //la puerta posee 4 vertices;indico los 3
        //y no indico el vertice pivote
        //diagrama puerta:
        //                       <-\
        //  pivote a1            a2     \
        //   *-----------------*      i
        //  i                  i    i
        //   *-----------------*
        //  b1                 b2
        //
        //movimiento lineal desde a2 a a2'(=lado
de puerta
        /*en vez de ser secto a aparte, es el  mismo
sector habitacion
        /con una pared oblicua que pongo derecha
al abrir la puerta*/
        //Vertice companero del pivote
        int a1;    //n de vertice
        //vertice en mismo lado que pivote
        int a2;
        //si es 0, noesta abierta, si es 1 esta abierta
        int o;
        //contador
        int i;
        int l; //numero de llaves requeridas
        //el tipo de puerta,  * = pivote
        /*                              ^         ^
           1    2       3    4      5  *----    7 ----*

           *    *       i    i      6  *----    8 ----*
           i    i   <- i    i ->        v          v
        <- i    i ->    i    i
           i    i       *    *

        */
        /*ademas, 4 tipos de puertas dobles

             a1 a2 b2 b1
                 ^              *   a1    *
           9 *---- ----*    11  i        i 12
                                i   a2   i
                               <i   b2   i >
          10 *---- ----*         i        i
                 v              i        i
                                *   b1   *
                                    */
        int t;
        //solo necesarios para las puertas dobles
        int b1;//segundo pivote
        int b2;//segundo extremo

        //una vez abierta, no se cierra
    END

    //tabla de sectores ascensor
    STRUCT ascensor[num_ascensores]
        int n=9999; //num sector
        int si; //altura de suelo inferior
        int ss; //altura de suelo superior
        int t;  //diferencia de altura techo-suelo
        int l; //numero de llaves requeridas
        int i; //para ver si esta bajado 0 o subido 30
    END

    //el teletransportador;
    STRUCT
teletransportador[num_teletransportadores]
        int n=9999;
        int b; //bandera a la que lleva.-1, otra fase
    END

    //para las fases
    int fase,fin_de_fase;
    //para mensajes
    string mensaje,mensaje2;
    string
nulo="ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
ÿÿÿÿ";
    int men;
    //para ver que manejo con la barra
espaciadora
    string usare;

    //manejo de FPGs
    fpg_wld,fpg_llaves;
    fpg_rombo,fpg_menu;
    fpg_sombra;
    numeros;
    //posicion del personaje
    int px,py,pz;
    int alt_suelo, alt_techo;

    int nnup,nfondo;
    int menu=0, opcion=1;
    //para on/off sonido, y volumen
    int on1,on2;
    int vol1;vol2;
    int volum1,volum2; //posicion del cursor de
volumen
    //posicion del modulo musical
    int posmus;
    //resolucion
    int res;
    //mapa de fondo entre fases para scroll
    int id_mapafondo, mp,mpup;
    //texto entre fases
    int id_texto_fase,texto_fase;
    //risa de EL MASTER
    int id_risa;

    //multiples vistas del ojo boby
```

```
void CMainFrame::
OnMenudatosbasicos()
    {
        CDatosPrincipal datos;
        datos.DoModal();
    }
```

Si compilamos ahora nos dirá que no conoce dicha clase. Para remediarlo, debemos incluir el archivo de cabecera que define la clase de ventana mediante la instrucción:

```
#include "DatosPrincipal.h"
```

Compilamos y ejecutamos, ahora ya podemos abrir la ventana de diálogo. Como vemos tan sólo hemos necesitado escribir un par de líneas y ya tenemos la estructura del programa.

Tan sólo tenemos que introducir la funcionalidad de los diversos controles, para eso, vamos a asociar cada uno de ellos a unas acciones determinadas. De esta forma, por ejemplo, conseguiremos mostrar el diálogo de abrir fichero para indicar el nombre del archivo FPG que deseamos cargar en nuestro programa.

Para crear el evento asociado a la pulsación de un botón, haremos un doble clic en el propio botón en ventana de diálogo. Se nos mostrará una ventana donde nos pide el nombre de la función asociada a dicho evento. El código que introduciremos será el siguiente:

```
void CDatosPrincipal::
OnButtoncargar()
    {
        UpdateData(TRUE);
        CFileDialog file(true,"fpg",
NULL,OFN_HIDEREADONLY|OFN_
OVERWRITEPROMPT,
            "Ficheros (*.fpg)|*.fpg|Todos
los archivos|*.*||",NULL);
        file.DoModal();
        m_FPG = file.GetFileName();
        UpdateData(FALSE);
    }
```

No vamos a entrar en detalles sobre los parámetros del constructor del diálogo de apertura. Tan sólo debemos saber que el primero nos indica que el diálogo es de apertura (si fuera de guardar, tendría el valor *false*). Lo que sí nos interesa es la asignación al nombre del fichero a la caja de edición mediante su variable asociada. Para que se muestre el valor obtenido del diálogo tras la asignación, debemos actualizar los datos mediante la función *UpdateData*. Nunca debemos olvidarla, ya que puede ser fuente de muchos errores.

Ahora ya tenemos todo lo necesario para crear nuestro programa de salida. Para poder obtener un resultado inmediato, vamos simplemente a desviar el fichero resultante al archivo *result.prg*, y para ello utilizaremos la salida estándar de C. Pero, ¿cuándo vamos a realizar dicha escritura? En este caso, vamos a escribir la salida cuando se pulse el botón de *OK* de nuestra ventana de diálogo. Para crear el evento asociado, basta con actuar como lo hicimos con el botón de cargar, e introducir el código que aparece en la tabla 2 en la nueva función:

Este código aunque aparentemente aparatoso, tan sólo se encarga de recoger los datos de la clase y darle el formato adecuado en forma de programa.

Conclusiones

Hemos creado lo que será la base de nuestro programa generador de código. Ya hemos obtenido algunos resultados y espero que os hayáis familiarizado un poco con el entorno de desarrollo. En el siguiente número realizaremos algunos cambios en nuestro programa, sobre todo en lo relativo a la creación de archivos de salida distintos de *result.prg* y la creación de nuevas opciones. Cualquier sugerencia acerca de opciones que se puedan añadir, creaciones propias, ideas, o lo que se os pueda ocurrir acerca de este pequeño programa, pueden remitirlas a la dirección *trinidad@arrakis.es*.

Pablo Trinidad

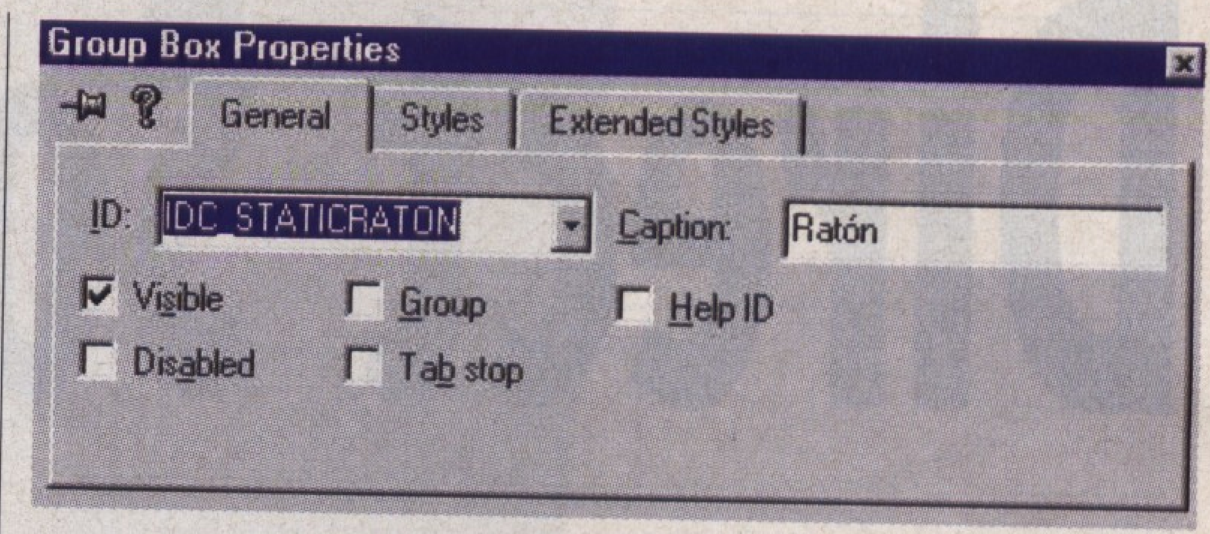


Figura 4. Ventana de propiedades del editor de recursos.

Tabla 2. Código

```
void CDatosPrincipal::OnOK()
{
    FILE *salida;

    UpdateData(TRUE);

    if ( (salida = fopen("result.prg","w")) == NULL)
        MessageBox("Error en la apertura del fichero de salida","Error");
    else
    {
        fprintf(salida,"PROGRAM %s;\n",m_NombrePrograma);
        fprintf(salida,"////////////////////////////////////////////\n");
        fprintf(salida,"// Autor: %s\n",m_Autor);
        fprintf(salida,"////////////////////////////////////////////\n");
        fprintf(salida,"\n\nBEGIN\n");
        CString res;
        if (m_ModoGrafico == "VGA Estándar - 320x200") res = "m320x200"; else
        if (m_ModoGrafico == "Modo X - 320x240") res = "m320x240"; else
        if (m_ModoGrafico == "Modo X - 320x400") res = "m320x400"; else
        if (m_ModoGrafico == "Modo X - 360x240") res = "m360x240"; else
        if (m_ModoGrafico == "Modo X - 360x360") res = "m360x360"; else
        if (m_ModoGrafico == "Modo X - 376x282") res = "m376x282"; else
        if (m_ModoGrafico == "SVGA VESA - 640x400") res = "m640x400"; else
        if (m_ModoGrafico == "SVGA VESA - 640x480") res = "m640x480"; else
        if (m_ModoGrafico == "SVGA VESA - 800x600") res = "m800x600"; else
        if (m_ModoGrafico == "SVGA VESA - 1024x768") res = "m1024x768";

        fprintf(salida,"set_mode(%s);\n",res);
        fprintf(salida,"load_fpg(\"%s\");\n",m_FPG);
        fprintf(salida,"\nput_screen(0,%d);\n",m_IDFondo);
        if (m_Raton)
            fprintf(salida,"Raton();\n\n");
        fprintf(salida,"LOOP\n\tFRAME;\nEND\n\nEND\n");
        if (m_Raton)
        {
            fprintf(salida,"\nPROCESS Raton()\n");
            fprintf(salida,"BEGIN\n\tgraph = %d;\n",m_IDRaton);
            fprintf(salida,"LOOP\n\tx = mouse.x;\n\ty =
mouse.y;\n\tFRAME;\nEND\nEND\n");
        }

        fclose(salida);
    }
    CDialog::OnOK();
}
```

Microsoft DIRECTX

Direct Sound v7

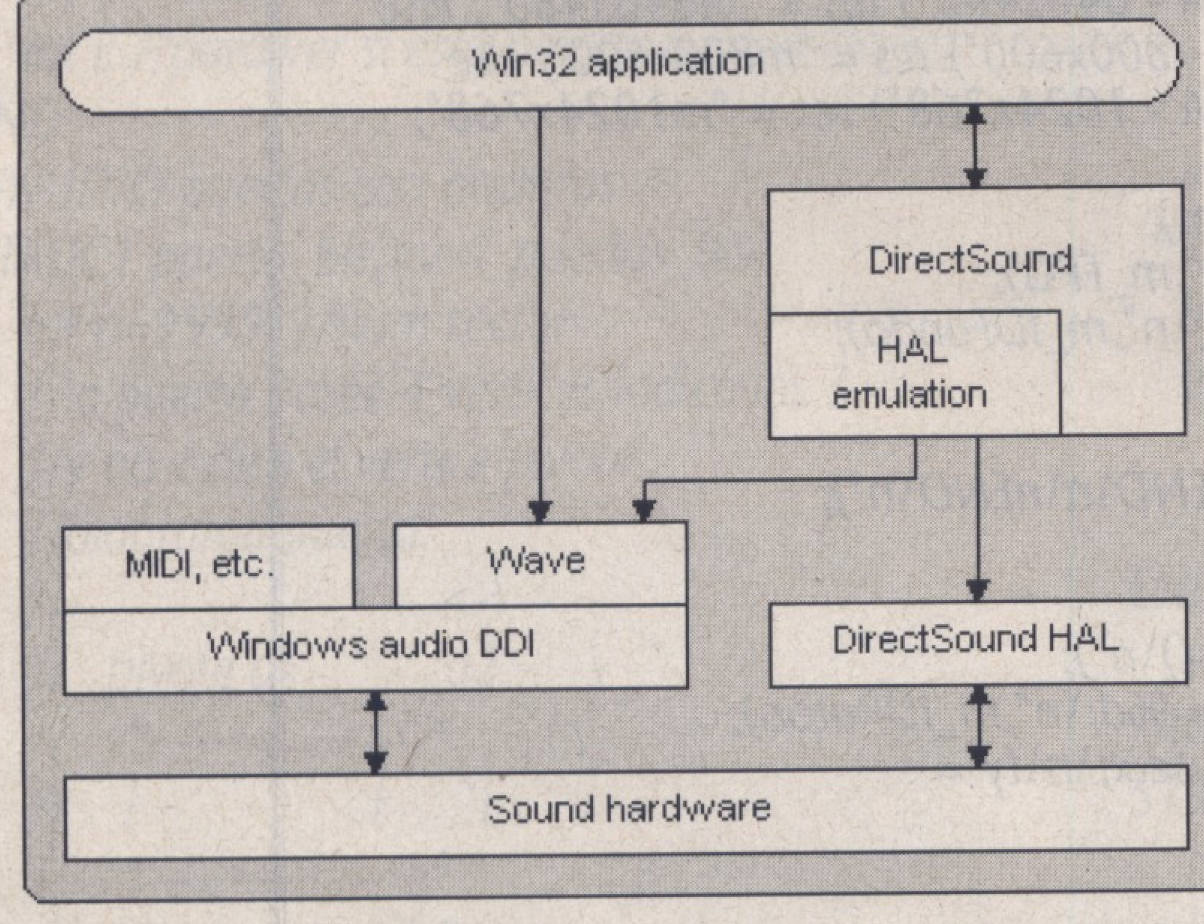
Tócala otra vez, Sam

Esta parte de DirectX se encarga de gestionar el sistema de sonido eficientemente. Lejos de aquellos tiempos en los que se utilizaba el lenguaje ensamblador para programar la tarjeta de sonido, esta API simplifica enormemente ese trabajo manteniendo una excelente eficiencia en sus procesos.

DirectSound supone un gran avance con respecto a los servicios MCI waveOut / waveIn de Windows e incluso da soporte al audio tridimensional, proporcionando un sistema de sonido impresionante.

Por si fuera poco, Microsoft ha comenzado su nueva versión de DirectX, la octava, que probablemente incluirá de una vez por todas un completo soporte para audio tridimensional- multicanal. Con el auge del DVD, y las funciones que nos brinda, el sonido estereofónico pasará a la historia en pro del sonido posicional, Dolby Digital AC-3, THX... y podremos disfrutarlo en casa, con ayuda de una tarjeta de sonido y unos altavoces adecuados, por poco precio. Los juegos y aplicaciones multimedia que hagan uso de estas tecnologías ganarán muchos enteros... ¡No esperemos ni un minuto más y veamos qué se esconde tras este Direct Sound!





Esta API traduce nuestras peticiones al driver de la tarjeta de sonido alejándonos de toda complicación a bajo nivel y acelerándolas por hardware cuando sea posible.

Arquitectura

El sonido se almacena en forma de buffer, de *sound buffers*. Son como archivos WAV que residen en una zona de memoria. Existen dos tipos:

- **Primario.** Aquí es donde se almacena la mezcla de los secundarios. Es el que va a parar directamente a la tarjeta de sonido (utilizando el DMA o bus PCI si el buffer se encuentra en memoria del sistema). Es único.
- **Secundario.** Almacena los datos para su mezcla en el primario. Puede estar en un formato diferente a éste y pueden ser varios.

Además, pueden ser tanto bidimensionales o tridimensionales. En este último caso es necesario establecer una "oreja" o *listener* en determinada posición para poderlos escuchar.

Dando el primer paso

Aunque el SDK de DirectX 7 viene preparado para trabajar con Visual Basic, aquí seguiremos la praxis habitual y utilizaremos C++.

Para empezar, es indispensable el include DSOUND.H y el lib DSOUND.LIB del SDK DirectX 7 en nuestro proyecto.

#include <dsound.h>

Lo primero es obtener una interfaz a un objeto IDirectSound mediante

```
LPDIRECTSOUNDlpDS;
DirectSoundCreate
(NULL, &lpDS, NULL );
```

El primer parámetro indica un GUID a un dispositivo específico. Si tuviéramos más de una tarjeta de sonido, deberíamos introducir un número de identificación para elegir una de ellas. De momento, seleccionaremos el dispositivo por defecto pasando un NULL.

El segundo parámetro es la dirección de la variable lpDS, donde se almacenará la instancia de DirectSound. Por último, volvemos a pasar un NULL en el último parámetro. En caso de error, lpDS se pondrá a cero y la función retornará un valor diferente a DS_OK. Esto suele suceder si no se tiene una versión correcta del mismo o se carece de tarjeta de sonido.

Ahora nos toca establecer el modo cooperativo, al igual que sucede con DirectDraw, DirectInput y casi todas las sub -APIS de DirectX. Para ello utilizamos:

```
lpDS -> SetCooperativeLevel
(hWnd, DSSCL_NORMAL);
lpDS -> SetCooperativeLevel
(hWnd, DSSCL_PRIORITY);
lpDS -> SetCooperativeLevel
(hWnd, DSSCL_EXCLUSIVE);
lpDS -> SetCooperativeLevel
(hWnd, DSSCL_WRITEPRIMARY);
```

Como podemos ver, el primer parámetro indica el *handle* de nuestra ventana. El segundo el modo de cooperación. Éste va desde normal (siempre a 8 bits, 22Khz) al manejo directo de la mezcla del primary buffer, pasando por los modos intermedios (priority en el caso de que nuestra aplicación consuma más recursos que las demás que estén sonando o exclusivo en el caso de que nuestro programa pueda silenciar al resto de aplicaciones y establecer el formato del primary buffer).

Como el fin de este artículo es introductorio, elegiremos el primero de ellos y nos limitaremos, por ahora, al sonido en dos dimensiones.

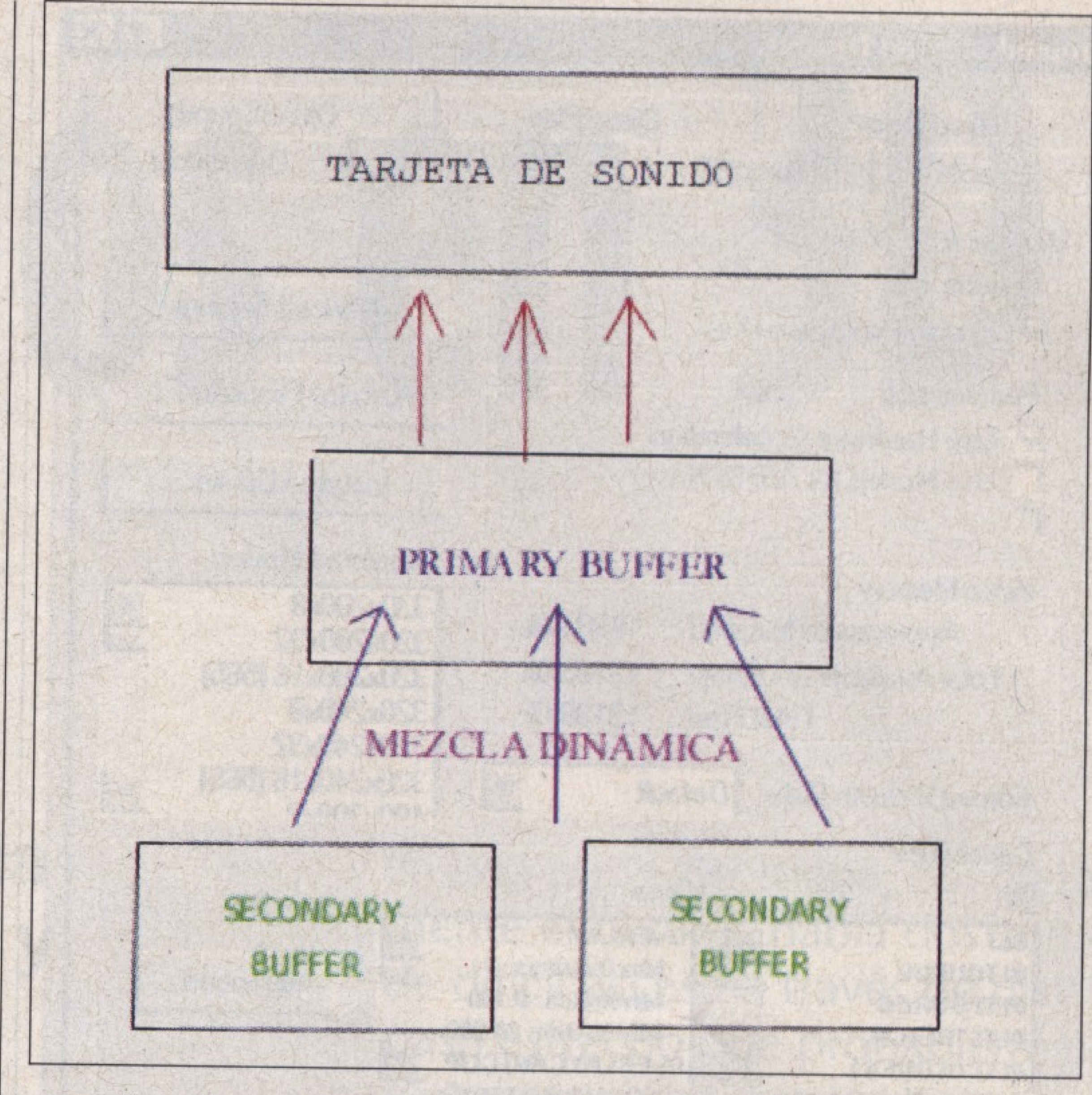
Creando el Buffer primario

Necesitamos rellenar una estructura DSBUFFERDESC de la siguiente forma:

```
LPDIRECTSOUNDBUFFER
lpPrimary;
DSBUFFERDESC        dsbd;

memset (&dsbd,0,sizeof (dsbd));
dsbd.dwSize = sizeof
( dsbd );
dsbd.dwFlags = DSBCAPS_
PRIMARYBUFFER;
lpDS -> CreateSoundBuffer
(&dsbd, &lpPrimary,
NULL);
```

Con lo que nos devolverá un puntero a un buffer primario si no hay ningún error.



Aquí podemos observar cómo los datos de los diferentes buffers secundarios se mezclan en el primario, que reside o se envía a la tarjeta de sonido.

Creando el Buffer secundario

Al igual que antes, rellenamos una estructura de forma parecida. A la hora de crear un buffer secundario, necesitamos especificar qué acciones se llevarán a cabo en él: cambio de volúmen, frecuencia, solicitud de posición actual, panorámica, así como el formato que utilizaremos. Si nos fijamos, al crear el primario no especificamos su formato. Esto es así porque al elegir un modo cooperativo DSSCL_NORMAL, no se nos permite controlar directamente el buffer de mezcla.

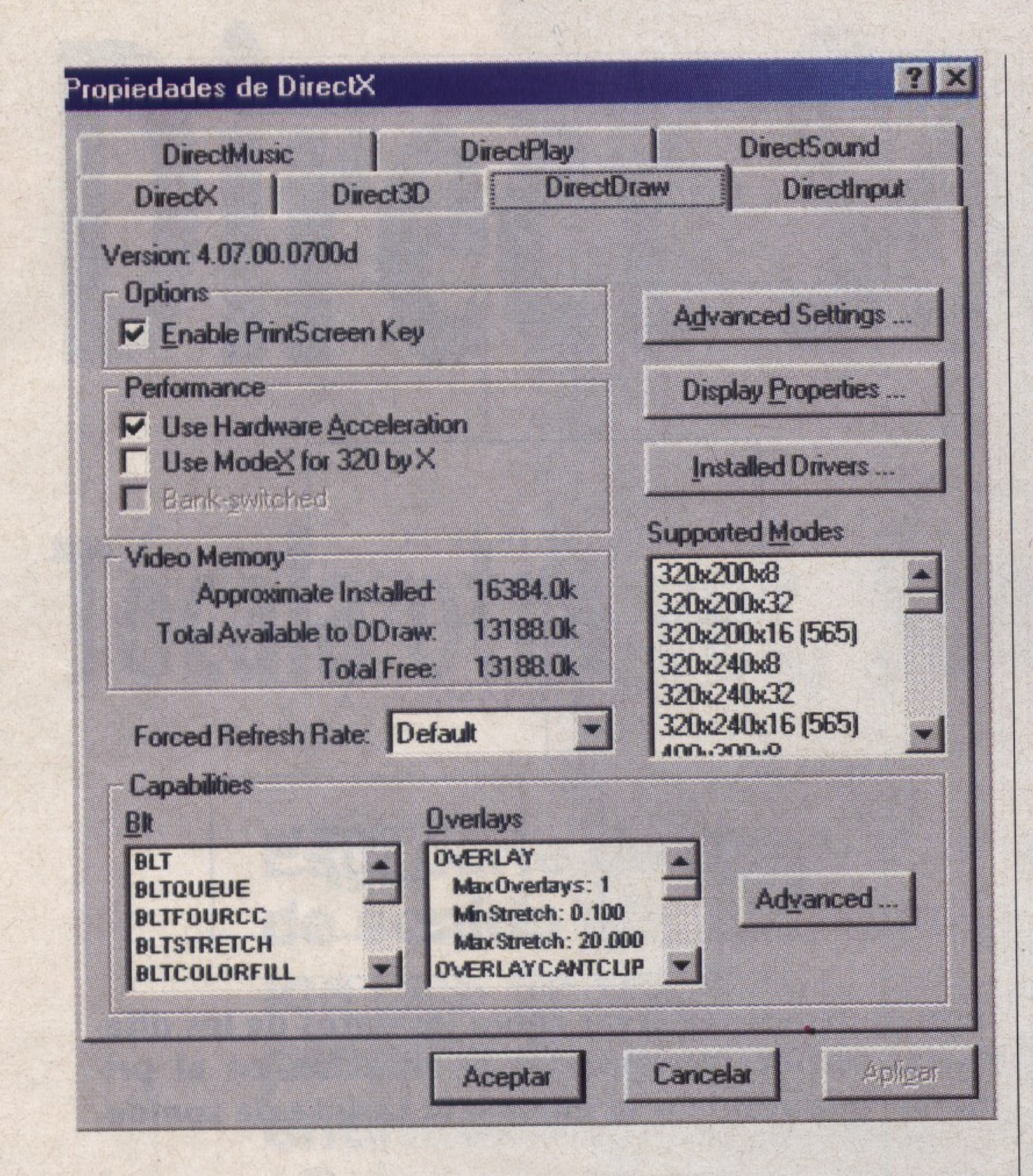
> A la hora de crear un buffer secundario, necesitaremos especificar qué acciones se llevarán a cabo en él

Bueno, aquí se especifica que queremos crear un buffer de 3 segundos a 22Khz con 16 bits de resolución Pulse Code Modulation (es decir, sin comprimir) con dos canales de audio (es decir, estéreo):

```
LPDIRECTSOUNDBUFFER
lpSecondary;
DSBUFFERDESC    dsbd;
WAVEFORMATEX    wf;

memset ( &wf, 0, sizeof( wf ) );
wf.wFormatTag = WAVE_FOR-
MAT_PCM;
wf.nChannels = 2;
wf.nSamplesPerSec = 22050;
wf.wBitsPerSample = 16;
wf.nBlockAlign = wf.nChannels *
wf.wBitsPerSample;
wf.nAvgBytesPerSec =
wf.nBlockAlign *
wf.nSamplesPerSec;

memset (&dsbd,0,sizeof (dsbd));
dsbd.dwSize = sizeof ( dsbd );
```



dsbd.dwFlags = DSBCAPS_CTRLPAN | DSBCAPS_CTRLVOLUME | DSBCAPS_CTRLFRECUENCY;
*dsdb.dwBufferBytes = 3 * wf.nAvgBytesPerSec;*
dsbd.pwfxFormat = &wf;
lpDS -> CreateSoundBuffer (&dsbd, &lpSecondary, NULL);

Rellenando el Buffer secundario

Ahora viene lo más difícil: hemos creado el buffer secundario, pero está vacío. Debemos rellenarlo con datos PCM (amplitudes en el eje Y, samples o muestras en el X). Para ello, nada mejor que *darle de comer* un archivo WAV - PCM (Direct Sound no admite compresión/descompresión directa). Para ello, debemos hacer un Lock del buffer (sí, seguro que esto te suena de DirectDraw), con lo que obtendremos un puntero void* en la memoria para poder trabajar con él:

Con el auge del DVD el sonido estereofónico pasará a la historia en favor del sonido posicional

void pointer;*
unsigned int bytes;

lpSecondary -> Lock (0, 0, &pointer, &bytes, 0, 0, DSBLOCK_ENTIREBUFFER);

El primer parámetro indica la posición desde la que se quiere comenzar a obtener el buffer. El segundo contiene el número de bytes que queremos obtener en caso que el primero no sea cero. Como tercer y cuarto parámetros introducimos el puntero donde obtener los datos así como un puntero a un *unsigned int* con los bytes que se bloquean. Acto seguido se indica otro puntero y cantidad de bytes bloqueados ya que los buffers secundarios son concebidos como circulares. Como nosotros estamos tratando con la totalidad del mismo, pasamos dos ceros y lo indicamos con el último parámetro.

Ahora debemos escribir los datos PCM en el puntero devuelto. El formato se adapta exactamente a los datos de un fichero WAV en su *chunk* "DATA". El problema está en que para hacer esto manualmente, se ha de conocer perfectamente la estructura interna de este tipo de archivos. Por suerte, Microsoft ha incluido en la API de Windows una función llamada mmioOpen (en mmsystem.h y winmm.lib) que nos puede facilitar mucho las cosas. También podemos echar un vistazo al archivo WAVREAD.C (situado en el directorio del SDK de DirectX 7 en \DSOUND\SRC\PLAYSOUND), con el que podemos cargar directamente un WAV en nuestro buffer.

En detalle:

HMMIO hmmio;
WAVEFORMATEX wf;
MMCKINFO mmckinfoData;

//Abro wav
WaveOpenFile ("miwav.wav", &hmmio, &pwfx, &mmckinfoParent);

//Busco datos PCM
WaveStartDataRead (&hmmio, &mmckinfoData, &mmckinfoParent);

//Aquí crearíamos el secondary buffer con el formato de la estructura pwfx y hacer Lock

//Leo datos
WaveReadFile (&hmmio, bytes, (BYTE)pointer, &mmckinfoData, &cbBytesRead);*

Una vez rellenado, desbloqueamos mediante

lpSecondary -> Unlock (pointer, bytes, 0, 0);

Con lo que los datos quedan guardados en la tarjeta de sonido o memoria del sistema.

Realizando acciones sobre Buffer secundario

¿Os acordáis de la bandera dsbd.dwFlags un poco más arriba? Dependiendo de lo que hayamos puesto, podremos controlar ciertos parámetros del buffer. Por ejemplo, si quitásemos el DSBCAPS_CTRLFRECUENCY, no podríamos cambiar la frecuencia de reproducción del buffer secundario.

Con lo que pusimos, podemos realizar las típicas funciones "VCR" sobre él:

lpSecondary -> Play (0, 0, DSBPLAY_LOOPING);
//Toca con loop
lpSecondary -> Stop ();
//Para la reproducción

Además, podemos variar la frecuencia de reproducción mediante:

lpSecondary -> SetFrecuency (30000);
//Pasa el sample a 33Khz, con lo que se hace más agudo

O cambiar la panorámica izquierda y derecha del buffer con

lpSecondary -> SetPan (-10.000);
//Aplica una atenuación de 100 dB al canal derecho

lpSecondary -> SetPan (10.000);
//Aplica una atenuación de 100 dB al canal izquierdo

lpSecondary -> SetPan (0);
//Centra la panorámica estéreo

También podemos atenuar el volumen original mediante

lpSecondary -> SetVolume (-10.000);
//Atenúa el volúmen en 10 dB

Conclusión

Bueno, espero que este modesto ejemplo haya servido para ver el funcionamiento general de DirectSound sobre el sonido en dos dimensiones: se inicializa, se crean y rellenan los buffers oportunos y se tocan o modifican sus propiedades. En próximas entregas explicaremos más extensamente el formato WAV y comentaremos algunas funciones 2D más avanzadas, indispensables para la gestión efectiva del audio en un juego o programa musical. Si tenéis alguna duda, queréis hacer algún comentario o simplemente intercambiar ideas, podéis escribirme un mail.

Santiago Orgaz
(santyhammer@latinmail.com)

Curso de juegos en 3D

DIV Deathmaker (IV)

En este número nos prepararemos para la batalla. Crearemos el Portal de la Muerte y permitiremos entrar en el mundo 3D.

Bien, parece ser que ya sólo quedaba meterse en la parte 3D y listo, pero como veis no es así. No os impacientéis; supongo que todos entenderéis que todas las partes de este juego son muy importantes y han ser explicadas detenidamente, además, la parte 3D y la IA de los bots en sí, posiblemente lleve varios números y no me gustaría mezclarlo con algo como el Portal de la Muerte.

Empezaremos con algunos cambios que ha sufrido nuestro programa.

Cambiando de look

No, no es un cambio drástico. Nuestro juego ha sufrido unas pequeñas modificaciones que no afectarán más que mínimamente al resultado final. Como todos sabéis es imposible diseñar el juego tal y como va a ser luego, dado que a favor de la optimización del programa hay que cambiar algunos métodos durante la etapa de programación. Div DeathMaker ha sufrido unos pequeños cambios de este tipo, pero que os explicaré con todo detalle para que no haya líos y os hagáis a la idea de lo que supone la modificación de los parámetros para ajustarse a la programación.

– La tabla WORD muerte[1]:
Teníamos tres tablas llamadas muerte[1] de tipo WORD, pero ahora no. Las tres globales, una se llamaba así y las otras dos estaban contenidas en las estructuras BOT[29] y PLAYER[7].
Ahora no hay WORD muerte[1]... hay dos INT por cada una de esas tablas, INT muerte0 e INT muerte1. ¿Por qué? Muy sencillo. Lo malo de usar tablas, especialmente no de tipo INT, es que surgen complicaciones a la hora de escribirlas en pantalla mediante WRITE o WRITE_INT (que no con write_in_map, pero lo consideré excesivamente complejo para un programa que pretende enseñar 3D y no programación, en general). Por ejemplo, no se puede calcular el OFFSET de los tipos WORD y BYTE dentro de una tabla... necesario para los WRITE_INT... y para los WRITE tendríamos que usar ITOA() que convierte una variable numérica a cadena de texto, pero surgen complicaciones al ser una tabla también, se requerirían strings temporales y eso es algo que ocuparía bastante más memoria puesto que tenemos 30 bots con 2 registros muerte[1] y en el Portal de la Muerte debe mostrar todos estos a la vez...
Por lo tanto, como dentro del juego debe mostrar también la puntuación de forma rápida, lo pasaremos a INT sin tabla para hacer WRITE rápidos.

– DEFAULT.WLD
El mapa 3D también ha sido modificado en parte para conseguir una mayor adaptación a la programación. Antes de llegar a la parte de la IA, no hay que ser muy avispado para darse cuenta de que no vamos a conseguir crear unos BOTs con una IA muy elevada y digna de un jugador humano, así que debemos "ponerle las cosas fáciles" al bot.

¿De verdad pensábais que programar era como escribir un cuento?

En una parte del mapa, concretamente el búnker, había dos zonas de difícil acceso a través de un paso estrecho que probablemente conseguirían que los bots entraran a recoger los ítems pero luego no pudieran salir. Esto ha sido corregido ampliando la entrada.

– DEFAULT.FPG
Lo admito, el FPG tenía unos fallos un poco gordos. El generador de sprites, al parecer, creó las imágenes demasiado oscuras y, al adaptarlas a la paleta, algunos colores se volvieron negro 0, es decir, negro transparente. Resultado: el personaje se volvía transparente en algunas zonas del cuerpo, a veces desaparecía el arma o algún brazo, etc. Esto ha sido solucionado; este FPG muestra perfectamente al personaje, tal como podéis ver en el gestor de bots al ver el skin.

Otra novedad

Si miráis el gestor de bots observaréis que hay un nuevo botón. Éste es un ojo y se encuentra al lado del botón de "cambiar ruta" en los bots. ¿Para qué sirve este botón? Muy sencillo. Es un botón de "PREVIEW". Cuando se pulsa este botón, se visualiza rotando el SKIN, corriendo en todas las direcciones. Para hacerse a la idea de cómo va a ser el bot. Se puede parar la visualización pulsando el botón derecho del ratón o alguna tecla. (Gracias a ManOwaR por convencerme finalmente de que lo pusiera).

Esta pequeña novedad se encuentra dentro de la función GESTORBOTS, en la parte del ELSE de IF (n==-1), esto es, cuando se trata de un bot y no del jugador.

```
IF (mouse.x>575 AND
mouse.x<592 AND mouse.y>84 AND
mouse.y<101);
   mouse.graph=996;
      IF (mouse.left AND raton==0);
      ve_skin(bot[n].skin);
      raton=1;
      END
END
```

Éste es el pequeño IF (sin comentarios, a diferencia del PRG) que se encarga de llamar a VE_SKIN(número de bot), función que se encarga de mostrarlo en pantalla. Básicamente, éste es el funcionamiento de VE_SKIN:

VE_SKIN tiene un parámetro de entrada, que es la cadena de texto que contiene la ruta del skin. Esta variable es de tipo PRIVATE y se llama fpg. También existe una variable PRIVATE llamada idfpg que nos servirá de ID para este fpg, para cargarlo y descargarlo de memoria.

Primero hay una serie de comprobaciones para que, en caso de que el FPG no se pueda leer o no exista, aparezca un mensaje de error y retorne con RETURN.

Suponiendo que se pudo cargar, sigue una serie de bucles que, cada uno, muestran el personaje corriendo en cada posición, detectando si se pulsó el botón derecho o tecla alguna para descargar el FPG y retornar. Todos ellos están integrados en un gran bucle LOOP para que, cuando termine el último bucle, vuelva al primero y se repita cíclicamente. El resultado, la preview que se puede apreciar al pulsar el ojo. La preview no era algo estrictamente necesario pero es de esos pequeños detalles que marcan la diferencia.

En nuestro programa, como en todo Deathmatch que se precie, habrá algo de sangre... pero sin pasarse

El portal de la muerte

Suena macabro, ¿verdad? Bien, de eso se trata. Después de todo parece ser que la sangre y la muerte son las dos constantes de todo Deathmatch que se precie (independientemente de que sea con bots o contra jugadores humanos). En nuestro juego habrá sangre, pero no en la medida que la hay en otros juegos en primera persona, aviso.

Si ejecutamos el programa, antes de mirar el código y elegimos *Empezar* en el menú principal nos encontraremos con un "Portal" que nos muestra los bots disponibles, sus puntuaciones, límites de frags y de tiempo, el WLD a cargar, y todo ello por encima de unas bonitas calaveras que hay de fondo (librería de Div), y un sugerente "Elige hasta 7 bots y prepárate para morir". A la izquierda de cada bot, una casilla para marcar los bots contra los que queramos jugar (recordando, hasta 7 bots) y abajo a la derecha, el incitante botón GO!; incitante porque incita a elegir rápidamente y probar, pero, como seguramente habréis comprobado ya, no permite mucho todavía. Aun así, si elegimos al menos un bot marcando su casilla con el botón izquierdo del ratón (podemos desmarcar todas las casillas con el botón derecho) y el mapa (que por defecto vendrá como "DEFAULT.WLD") veremos como aparecen textos en pantalla como "Cargando mapa de modo8..." o "Cargando SKINs, llamando BOTs...", que se corresponden con la realidad, es decir, lo que está haciendo el programa en ese momento y si no surge ningún problema de memoria (yo tengo 16mb de RAM y de momento, funciona, aunque claro, sólo uso un SKIN: Default ;) iniciará el mapa, y veremos el mapa seleccionado ante nosotros, por el cual nos podremos mover e incluso ver a los bots, que estarán quietecitos.

Pasemos a analizar la función portalmuerte() en profundidad. Las variables PRIVATE ya están explicadas en el PRG, así que pasamos directamente a las instrucciones.

Lo primero que nos encontramos es un "ignore_error(159);". Esto ignorará los errores de tipo "no se pudo encontrar el WLD", es decir, cuando no se pudo cargar el archivo WLD especificado. Si no se hubiera puesto, al escribir mal el nombre del WLD, por ejemplo, se saldría del programa diciendo "Error 159: no se pudo encontrar el WLD". De este modo, lo único que pasará es que no se verá nada. Pero al pulsar ESC volveremos al menú. Pretendía haber incluido un mensaje "ERROR: WLD no encontrado" con retorno al menú pero no se puede hacer o, al menos, no de una forma sencilla (sin apaños/chapucillas:), ya que para un FPG sólo hay que realizar una pequeña comprobación tipo IF (load_fpg(loquesea)==0) ya que load_fpg retorna 0 si no se pudo cargar, pero load_wld (como pude comprobar) retorna siempre 0, y esto es en parte lógico. ¿Por qué? Sencillo. Load_fpg retorna la ID del fpg; si no se ha cargado ésta es 0. Pero como sólo se puede cargar un WLD a la vez, y al cargar uno se descarga de memoria el anterior retornar la ID es una tontería, por lo cual no lo hace. Y no se puede comprobar si se cargó bien o no (no de esta forma).

Una pequeña nota, los ignore_error deberían ir todos juntos al principio del programa para, con un vistazo rápido, saber qué tipo de errores se están ignorando, pero están situados expresamente en las funciones donde más repercuten para guiaros.

Tras eso hay que resetear las variables "selected[7]" (salvo selected[7] que la aprovechamos para el



Algunos conocidos esperan en el Portal...

ratón), en las que se contienen los números de los bots seleccionados, para que al salir de una partida o del Portal y volver a entrar no sigan seleccionados los mismos.

Después, el bucle <<FROM n=0 TO 29; bot[n].puntos=bot[n].muerte0-bot[n].muerte1>> se encarga de calcular la variable puntos de cada registro de bot[29], a base de restarle a los asesinatos las muertes, para mostrarlo luego en la lista de bots. Y un poco más abajo otro bucle FROM se encarga de escribir todas las puntuaciones y relaciones de asesinatos/muertes. En azul, asesinatos, en rojo, las muertes, y en verde, la puntuación neta. Y una nota: este FROM escribe las puntuaciones sólo cuando el bot correspondiente tiene tanto nombre como un skin definido, si no, escribe dos guiones en el lugar del nombre "- -".

Unas pocas líneas más abajo nos encontramos con las llamadas a los procesos "marca".

Al pulsar en una casilla (de un bot seleccionable) debe aparecer una cruz, que también debe poder moverse al borrarla y marcarla en otro sitio. Cada proceso marca corresponde a un número de bot (el primero elegido, el segundo,...) y cada una tiene su registro de selected[7] para comprobar cuándo debe situarse en una casilla al pulsarse el ratón sobre una. El proceso marca es muy simple y fácilmente entendible; de todos modos, está comentado en el PRG así que no es necesario parar sobre eso.

A continuación viene un bucle REPEAT. Es el bucle que se encarga de manejar en sí el Portal de la Muerte, aunque, como veréis, esta función no sólo controla esa pantalla. El bucle no es que sea muy simple pero no tiene nada especialmente complicado; es muy similar al bucle de gestorbots que se encargaba de "hacer funcionar" los botones, por lo que sería una repetición comentarlo otra vez. Sin embargo, hay un par de cosas a destacar: primero, que no hace un delete_text(all_text) tras cada FRAME



El bot ManOwaR se acercó tanto que se puso a pixelar.

porque eso borraría también todas las puntuaciones, y no nos interesa porque no van a ser cambiadas en ese momento y son muchas como para reescribirlas a cada FRAME. Sólo hay delete_text de los textos modificables (límite de frags y tiempo y ruta y nombre del WLD) para poder ser actualizados; segundo, si se pulsa ESC no hay más que hacer que borrar todo de pantalla (y borrar y descargar todos los textos e indicar con un estado=0 que se vuelve al menú), siguiéndole un RETURN porque el REPEAT no acaba cuando se pulsa ESC (como gestorbots) sino cuando se hace clic en GO! y hay, al menos, un bot elegido. Por ello, la comprobación de si se pulsa ESC está dentro de un IF en el bucle.

Cuando se pulsa en GO! se borran los textos y la pantalla, se indica que estado=3 (en proceso de carga, aunque este valor de estado no es especialmente útil) y comienza todo el susodicho proceso de carga de los recursos del juego.

Esta es una parte delicada y bastante importante. Si no se realiza con mucho cuidado luego pueden surgir problemas en ordenadores con poca memoria y, además, en todos los ordenadores, podemos estar desperdiciando memoria, y eso es un lujo que no nos podemos permitir. Como podéis ver en el proceso de carga, hay una parte rebosante de IFs muy complicada, y como ya aparece en los comentarios del PRG, es algo que en un principio no iba a incluir pero era algo tan importante que al final lo metí. Es una parte del código compleja que no es necesario que comprendáis, lo que hace es lo siguiente:

Supongamos que todos los bots (los 7 que llamemos) tienen de skin "default". Sería un desperdicio de memoria tener que cargar 7 veces los fpgs, una para cada proceso. Se encarga de mirar si ya se había cargado un skin para no tener que cargarlo otra vez.

Pero cuidado, si un bot tiene de ruta de skin "default.fpg" y otro "c:\div2\default.fpg", aunque sea al fin y al cabo el mismo directorio, al ser distinta la cadena de texto DDM los cargará por separado. Igual que si uno es "default.fpg" y otro"default.fpg " con un espacio detrás, son distintas así que se consideran dos skins distintos.

El resto del proceso de carga es muy sencillo, hasta que empieza a mirar qué bots se han elegido y se prepara toda la estructura PLAYER[7], se llaman a los procesos botp() y a jugador(), y se inicia el modo8.

Y aquí viene otra parte nueva. Se resetea la variable global STARTSPOTS, poniéndola a 0. Se ignora el error 156, tipo "el objeto se encuentra fuera del mapa y será eliminado" para que al mandar los siguientes procesos sonda(), si no se encuentra dentro del mapa (va a una bandera inexistente, en otras palabras) no diga nada, y lo mismo por si por algún casual algún bot desaparece por esa causa (que sólo puede ocurrir si se usa un mapa que no sigue las reglas de las posiciones de inicio, explicadas más abajo).

Los procesos sonda son muy simples. Sólo van a la bandera indicada, hacen un FRAME y en la siguiente instrucción suman 1 a STARTSPOTS. Si la bandera no existía nada más hacer el FRAME el proceso hubiera desaparecido, por encontrarse fuera del mapa. Por lo que no hay que hacer ni comprobación ni nada. Al final STARTSPOTS contendrá el número de lugares de comienzo a partir de la bandera 1. Útil para cuando, al aparecer un bot/jugador, se use go_to_flag(rand(1,startspots)) lo que le llevará a una bandera al azar entre 1 y el número de banderas de comienzo.

Justo después se comprueba si es necesario inicializar el TIMER[9] para llevar el límite de tiempo, y se entra en el LOOP que sólo se rompe mediante BREAK; (continúa tras el LOOP) al alcanzar un límite o con RETURN al pulsar ESC (con la consiguiente pérdida de las puntuaciones de la partida).

Tras alcanzar un límite, después del LOOP está una pequeña sección de código que simplemente muestra los resultados de la partida sobre un fondo negro. Hay que decir que falta una cosa al final de esta sección, que guardase las puntuaciones obtenidas en cada bot y el jugador y reordenase los bots. Pero eso, en el siguiente número cuando ya se puede "andar" en condiciones por el mundo 3D.

Quiero jugar

Como sabéis, jugar, lo que es jugar, no se puede todavía. En realidad todo lo que quería incluir era hasta el momento en que se ha cargado todo para empezar a jugar y se ha preparado la estructura player[7], pero sabía que cargar todo y que se quedara paralizado ahí era un poco frustante. Así que incluí los procesos botp() que controla a los bots y jugador(), que controla al jugador como su nombre indica, con unos controles temporales para que se pudiera andar por el mapa. Esta parte del código todavía no estaba pensada para ser comentada aquí y parte de ella es provisional, así que no es necesario ni entenderla ni mirarla siquiera. En el siguiente número hablaremos a fondo de estos dos procesos, así que, no os impacientéis.

Para que os controléis un poco por el mapa y os hagáis a la idea de los controles, éstos son:

- Flechas arriba-abajo: hacen al jugador avanzar o retroceder.
- Flechas izquierda-derecha: hacen strafe-left y strafe-right. En términos no-quakeros, desplazan lateralmente al jugador hacia la izquierda y la derecha.
- Ratón: controla la vista.
- Tecla Q o botón derecho del ratón: hace "volar" al jugador. No os paséis con esto, lo he puesto para que "naveguéis" libremente por el paisaje, pero sabéis que el modo8 no es perfecto y a mucha altura y al visualizar muchos sectores a la vez... (no me responsabilizo de los bloqueos) aunque es verdad que he registrado poquísimos bloqueos cuando estuve probando bugs de este mapa.
- Tecla A: hace descender al jugador más rápido.

Jugar, lo que se dice jugar, aún no se puede, pero la cosa ya va bastante avanzada

Una curiosidad

Si sois de esos que buscan fallos por todos sitios os habréis dado cuenta de un fallo que, aunque no llega a ser un fallo propiamente dicho es, cuando menos, una pequeña curiosidad.

Imaginaos que tenéis tres bots, de puntos (frags, puntuación neta) 3, 2 y 1. Vais al gestor de bots y al segundo le borráis el nombre o la ruta del skin, o ambas cosas. Al pulsar ESC se reordenan los bots por orden de puntuación, de mayor a menor, y el segundo bot, aún sin nombre ni ruta de skin permanece el segundo.

Ahora vais al Portal de la Muerte y... ¡sorpresa! Como el bot 2 no tiene

nombre ni ruta del skin (o alguna de las dos) no aparece, aunque hay un bot no utilizable entre el bot 1 (3 puntos) y el 3 (1 punto). Se supone que tendrían que aparecer en orden pero hay uno no utilizable en medio. ¿Es un fallo? Bueno, no es algo realmente grave. Está bien desde el punto de vista de que "avisa" al jugador de que tiene un bot con puntuación que no tiene nombre o skin por lo que no puede usarlo.

En teoría, al borrar un bot se debe borrar manualmente el nombre y el skin y resetear las puntuaciones. Los datos de agresividad, agilidad, etc., dan igual.

Pero es inútil borrar un bot, en todo caso, sería sobrescribirlo. Porque ocupa tanto un registro de bot ocupado como uno vacío, en BOTS.CFG. Y aunque no se use ese bot, no viene mal tener uno más.

Un par de consejillos

Hete aquí un par de consejos para que nos resulte más fácil ejecutar DDM desde el entorno de Div hasta que podamos compilarlo al final.

Lo primero, recordad que necesitamos un BOTS.CFG válido; Div 2 lo incluye, al crear la instalación, en el programa, pero mientras se hacen las pruebas y el programa se ejecuta desde el entorno de Div es el programador el que tiene que crear este archivo; tanto en nuestro programa como en todos. No es muy difícil, sólo hay que ejecutarlo una vez cambiando los FREAD (cuando carga los datos al principio) por FWRITE, y modificando el FOPEN como algún tipo de los que crea el archivo en caso de no existir, por ejemplo "w+" (lo crea y si ya existe lo trunca). Ejecutando, ya tendremos el BOTS.CFG en el disco duro, salimos, restauramos los FREAD y FOPEN y ya está.

Hay un nuevo punto que no podemos saltar a la hora de hacer un mapa para Div Deathmaker

Pero esta vez no hará falta que lo hagáis; en el CD encontraréis BOTS.CFG ;) copiadlo a vuestro directorio de Div 2 y no habrá problemas.

Lo segundo, recordad que para Div 2, al cargar un archivo, si no se especifica ruta, la ruta por defecto es vuestro directorio de Div 2. Si no queréis tener que escribir en las rutas del skin/mapa "C:\PROGRAMAS\lenguajes\programacion\div\div2\dd\default.fpg" por ejemplo, copiad DEFAULT.WLD y DEFAULT.FPG a vuestro directorio de Div 2, con lo que sólo necesitaréis poner "default.fpg" en la ruta del skin, por ejemplo, para que lo cargue.



Qué quietecito está... en un par de números más adelante y desearemos que nos dé un respiro.

Nota: en el juego, una vez compilado, el directorio por defecto donde buscará, será, por supuesto, el mismo del juego. Y tanto DEFAULT.FPG como DEFAULT.WLD estarán en el mismo directorio del juego.

Y nota 2: Div 2 no incluirá DEFAULT.WLD ni DEFAULT.FPG en el PAK, ya que los load (de lo que sea) que realicemos de una cadena de texto (como el load_fpg(bot[n].skin), al suponer que son cadenas de texto variables, Div 2 los considera procesos de carga de archivos externos que pueden ser definidos por el usuario, por lo que es imposible meterlos en el PAK antes de tenerlo ni saber dónde está. Así que una vez compilado el programa y hecha la instalación, hay que incluir DEFAULT.WLD y DEFAULT.FPG, así como los TXT pertinentes y todo lo que se quiera añadir. Y como no podremos incluirlo en la instalación, habrá que instalarlo y crearnos nuestra propia instalación con los archivos que necesitemos (y eso pasa en todos los juegos que necesitan de algún archivo extra que no incluye Div 2 en la instalación).

En en CD

En este número, en el directorio de la sección 3D/RED se encuentran:

- Todos los archivos ya disponibles en el anterior número, teniendo en cuenta que:
  - DD.PRG ha sido actualizado, por supuesto.
  - MISC.FPG ha sido ampliado; ahora incluye la pantalla del Portal de la Muerte y algún gráfico como el del puntero del ratón en modo "Preview".
- Y, además:
  - BOTS.CFG: es el archivo donde Div DeathMaker guarda la configuración y los datos de los bots. Es un archivo que necesita para escribir en él los bots. Si lo guardáis en el directorio de Div 2 no habrá problema. Al crear la instalación Div 2 se encargará automáticamente de crearlo con los valores predeterminados (0 todos en nuestro juego) y meterlo en nuestro juego.

Creación de mapas para DDM

Hay un nuevo punto a tener en cuenta a la hora de crear un mapa para Div Deathmaker, que hasta ahora (perdonadme) no había advertido.

Es posible establecer desde 1 a 10 banderas para las posiciones de inicio, como ya dije unos meses atrás, pero deben empezar en la bandera número 1 y ser una sucesión continua, en otras palabras, si sólo va a disponer de 4 banderas el mapa, deben ser las banderas 1,2,3 y 4. Pero nunca la 1, la 5, la 6, la 8 y la 9 por ejemplo. Ni tampoco la 2, 3, 4 y 5.

¿Por qué? Porque DDM tiene que saber cuáles, de los lugares de comienzo, van a estar disponibles, porque si mandamos a un proceso botp() a una bandera inexistente, desaparecerá. Existe una instrucción ignore_error(156) (explicada antes) que ignora los errores tipo "El proceso se encuentra fuera del mapa y será eliminado" así que desaparecería sin más. Y para asignarle la bandera de comienzo a un bot/jugador DDM usa la función rand; rand requiere de un intervalo para sacar el número al azar y no podemos usar valores alternados, por poderse hacer, os aseguro que se puede hacer (y de hecho si el juego no lo hiciera con fin didáctico yo lo hubiera hecho) que no haya problema de elegir las banderas que se quieran del 1 al 10 pero lo veo una complicación innecesaria. El que quiera crear un mapa 3D para DDM tendrá que ajustarse a esta regla.

Y para concluir, desafortunadamente sólo se pueden crear mapas para DDM con Div 2 y su editor de mapas WLD. Pero es posible que en breve un divero de la comunidad Div consiga hacer un editor de WLDs que no requiera Div, ¡desde aquí todo mi apoyo!

Hasta la vista baby

Se acabó este número; no os olvidéis de mirar el PRG para todo aquello que no entendáis bien. Espero que no os haya sabido a poco, ya sé que muchos lo que queréis es poder jugar, pero dadle tiempo al tiempo

Recordad que si tenéis alguna sugerencia/duda/o lo que sea me lo podéis mandar a mi e-mail, las 24 horas del día los 365 días del año.

¡Hasta el próximo número!

Ferminho (Fermín Vicente)
ferminho@lycosmail.com

Programación de Juegos de Estrategia - 8

Principios básicos de IA

Como ya adelantamos en el número 7 de la revista, en esta ocasión vamos adentrarnos en la parte más importante de un juego de estrategia. Veremos los principios básicos de la inteligencia artificial o, lo que es lo mismo, simulación del comportamiento humano por parte de la máquina. La vamos a dividir en dos partes: la que controla las reacciones de las unidades ante ciertos estímulos y la que coordina las unidades para que actúen como un ejército.

Empezaremos definiendo claramente nuestro objetivo. Lo que pretendemos hacer es dotar a nuestras criaturas de unas pautas de comportamiento para que el ordenador simule ser un adversario inteligente. En el fondo, el comportamiento, tanto de jugadores controlados por humanos como el de los que son controlados por la máquina, se basa en pares acción-reacción. En términos informáticos, sería una sentencia *if* (acción) *then* (reacción). Para organizar estos pares, se agrupan en transiciones entre diferentes estados. Como el concepto es algo confuso lo explicaremos con un poco más de detalle.

PARADO
ATACK
EXPLORE
HUYENDO

Aquí vemos un grafo de estados de ejemplo.

Grafos de estados

Los programadores menos experimentados tal vez no sepan lo que es un grafo, por lo que vamos a explicarlo. Un grafo es algo tan simple como un grupo de nodos interconectados por aristas, como podría ser un mapa de carreteras (los nodos serían las ciudades y las aristas las carreteras). La utilidad de los grafos es modelizar situaciones de la vida cotidiana como el ejemplo dado del mapa de carreteras. En artículos anteriores, se mencionaba el algoritmo Explora-grafos como solución optima de búsqueda de caminos. En él se supone cada casilla como un nodo, que estaría interconectado con las casillas adyacentes a las que se pudiera acceder. En nuestro caso, los nodos serán estados de un *mob* (andando, quieto, huyendo...) y las aristas serán transiciones entre ellos.

Gracias a este modelo si un *mob* en un estado recibe un estimulo (acción), se produce un cambio a otro estado (reacción) que implica un comportamiento diferente. Por ejemplo, si un *mob* está quieto y ve un enemigo, podría pasar al estado de "atacando" o al estado de "huyendo" (según la unidad). Como se puede comprobar, este modelo simplifica mucho el diseño de la IA de nuestros *mobs*, ya que una vez que decidamos todos los posibles estados en los que puede estar (no suelen ser muchos aunque en un principio lo parezca) sólo debemos pensar qué haríamos nosotros en esa situación. De todas formas, queda ver cómo programar este modelo que, ahora lo veremos, no es tan complicado.

Implementación de un grafo de estados

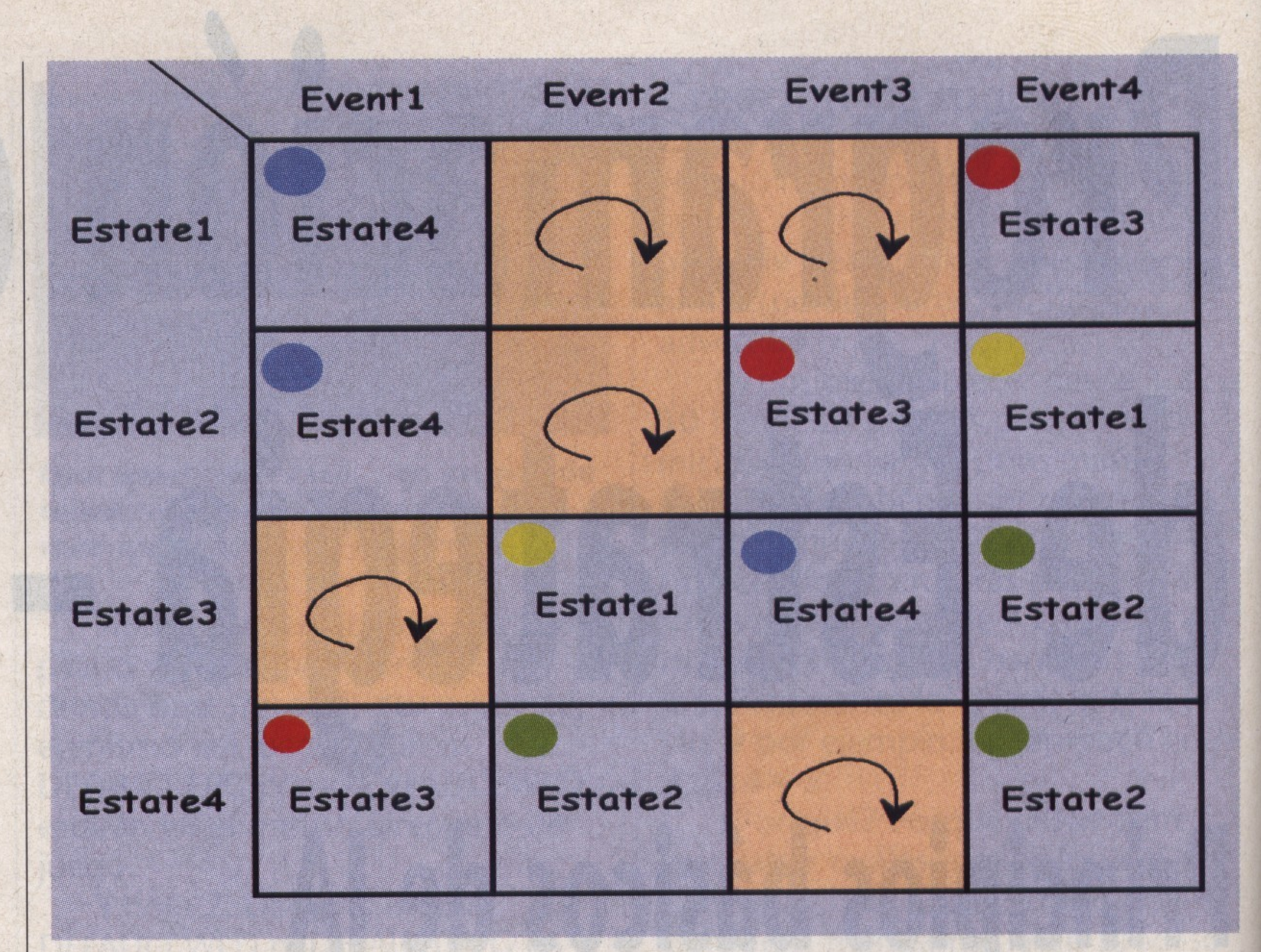
A algunos, esto os recordará un poco a los autómatas deterministas y, a otros, las máquinas de estados de Moore y Mealy. En el fondo así es, por lo que su programación será similar. En cada interacción del bucle del proceso *mob* comprobaremos en qué estado nos encontramos, lo que nos situará en una determinada región de código, en la que se comprobará si se cumple alguna de las condiciones que provocaría una transición a otro estado, además de realizar las acciones asociadas a ese determinado estado. En muchos casos, podrán mantenerse las transiciones entre estados en una matriz en la que las filas

sean el estado origen, las columnas sean las acciones (habría que asociar un número a cada situación) y el contenido sea el estado al que se pasaría cuando en la fila se produjera la acción representada por la columna. Es posible que para realizar las acciones asociadas a un estado se necesite información adicional. Por ejemplo, para poder atacar, debemos conocer el objetivo de nuestro ataque, al igual que para movernos debemos conocer las coordenadas del destino. Esta información se obtiene en el estado desde el que se llegó a éste. Es decir, si estamos en el estado "quieto" y cuando vemos a un *mob* enemigo pasamos a "atacando", en el estado "quieto" deberemos indicar también que el objetivo del ataque es el enemigo avistado. Esto es así porque a un estado se puede haber llegado de muchas maneras y no saber desde qué estado ni por qué. En cambio, sí qué sabemos a qué estado vamos cuando salimos de uno y la razón por la que salimos, por eso podemos indicarle lo que queramos.

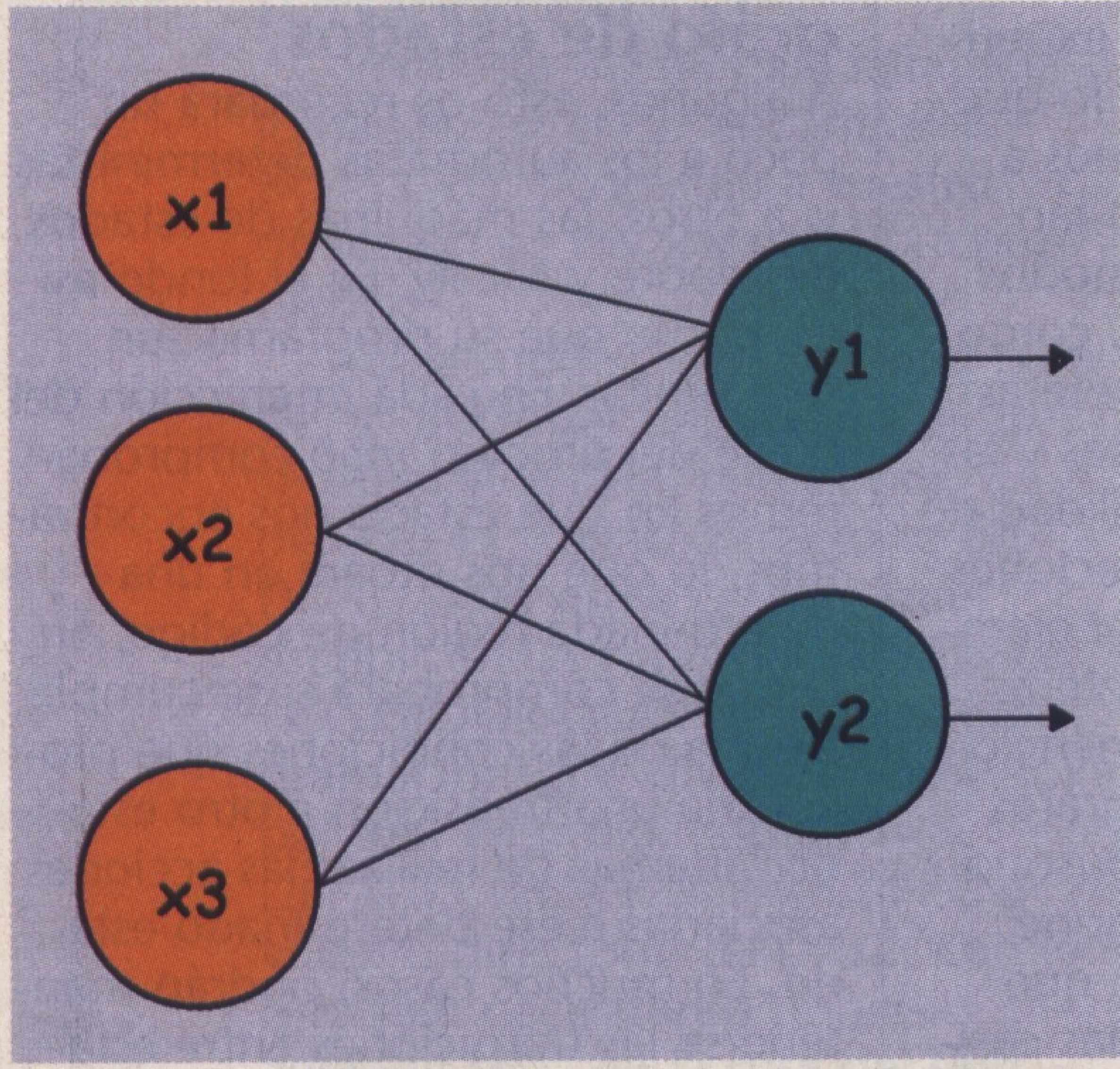
En muchos casos podrán mantenerse las transiciones entre estados en una matriz

Controlador general

Hasta el momento hemos visto el comportamiento individual de los *mobs*, sin tener en cuenta el estado general del bando completo. Se podría decir que tenemos controlado el comportamiento instintivo de las unidades, pero no el comportamiento racional. Este comportamiento debe ser controlado por un proceso independiente que simularía las acciones que lleva a cabo el jugador humano dando órdenes a sus unidades. Este proceso no tendría en cuenta las acciones sobre una unidad determinada, sino que comprobaría qué acciones son necesarias para la supervivencia y victoria de todo el bando. Esto implica comprobar el estado de los recursos del bando, el número de unidades, la localización del enemigo, etc. Además, se encarga de organizar las estrategias a seguir en función de la información disponible. Lo más normal es programar un número de objetivos básicos ordenados por orden de prioridad. Por ejemplo, se intentará crear un número prefijado de peones y militares, determinados edificios, etc., para tener una estabilidad antes de empezar la ofensiva. Luego se mandaría a un número de unidades a explorar el terreno en busca de enemigos, comprobar el tipo de defensa que posee el enemigo y crear unidades específicas para contrarrestar esa defensa. También se pueden programar varios tipos de comportamiento, como podría ser una actitud principalmente destructiva o defensiva de desgaste (acaparar todos los recursos hasta que se agoten y luego realizar una ofensiva definitiva). Esto entra ya en el apartado de estrategias y cada uno programará las que mejor le parezcan, pero el sistema es el mismo: decidir una serie de objetivos a cumplir, en un orden y hacer que los *mobs* los realicen. Esto es un sistema de control estático que prácticamente no varía según el estilo de juego del jugador humano. Los sistemas de inteligencia artificial adaptativa son sistemas complejos en los que el ordenador aprende de sus errores y de sus éxitos, adaptando su comportamiento posterior a lo aprendido anteriormente. Un ejemplo de estos sistemas son las redes neuronales. En este curso no vamos a adentrarnos en estos temas, pero lo mencionamos para que sepáis qué buscar si deseáis ampliar conocimientos sobre IA. El proceso controlador, más complejo que el instintivo, será el que iremos desarrollando en próximos artículos.



Ésta es una matriz de estado típica (la flecha indica que no hay transición).



Perceptrón (el tipo más común de red neuronal).

Para terminar

En el próximo número empezaremos el diseño de nuestro controlador general así que no os lo perdáis.

Emilio Llamas Alba (YuMoK)

Pseudocódigo relativo al control de estados

```
LOOP
.
.
.
Switch(estado_mob)
{
   case PARADO:
      //comprobacion de
transiciones
      if(casi_muerto())
{
estado_mob=HUYENDO;
destino=home();
      }
      //acciones del estado PARADO
   case MOVIENDO:
      //comprobacion de
transiciones
      //acciones del estado
MOVIENDO
}
.
.
.
END
```



El Francotirador: BattleZone

BattleZone es un adictivo juego de estrategia en tiempo real, con el aliciente de ser un juego en 3D con vista en primera persona. Lo más notorio de este juego es su uso de las 3D, introduciendo al jugador mucho más en la acción. Veamos qué técnica utiliza.

'Voxels'

El terreno de juego nos recuerda al de otros juegos como *Comanche* y otros de aviación, ya que todos utilizan una representación del terreno con *voxels*. En el fondo un voxel es como un píxel, sólo que, en vez de un color, representa una altura. Se tiene un mapa del terreno en el que cada punto representa una altura, y se pintan aristas entre puntos colindantes. El resultado es una malla tridimensional que, al aplicarle textura, y, opcionalmente, sombreado *goraud* (ajuste de la intensidad de las texturas en la zona de las aristas), queda muy realista. El problema, al igual que pasaba con DOOM y con el sistema 3D del DIV2, es que en un punto (x,y) solo puede existir una altura, por lo que no se pueden hacer edificios de varias plantas, ni túneles, ni nada por el estilo. Los objetos, en cambio, están realizados en 3D real, lo que le da un toque mucho más realista. Aun así, se echan en falta más polígonos en los objetos pero lo que se pierde en realismo se gana en velocidad.

Sistema de juego (interfaz)

A diferencia de los juegos de estrategia en 2D, en *BattleZone* el jugador toma posesión de una identidad en el juego, por lo que, además de organizar la estrategia, participa activamente en ella haciendo las partidas mucho más adictivas. Las órdenes estratégicas, como pueden ser la construcción de unidades o el movimiento de éstas por el mapa, se realizan por menús organizados según el tipo de unidades, de acciones, etc. Cuando situamos una unidad en el mapa, vemos que sólo puede situarse en ciertas zonas como hasta ahora, es decir, que también utiliza una rejilla para controlar el juego.

La IA

La IA del *BattleZone*, aunque resulte difícil de creer en un juego 3D, no es ni más ni menos compleja de programar que en un juego 2D, ya que, en el fondo, lo único que cambia es la forma de representar la información. Al igual que en otros juegos, todas las unidades están encasilladas en una rejilla sobre la que se hacen los cálculos. La única diferencia es la variedad de alturas diferentes, lo que permite moverse de una casilla a otra en un sentido, pero en el contrario no. Por ejemplo, se consideraría inaccesible una casilla (B) cuya altura con respecto a la actual (A) fuera mayor que *H_MAX*, pero si estamos en la casilla B y queremos ir a A podemos dejarnos caer.

Despedida

Nos despedimos comentando que ya existe *BattleZone II*, con objetos más detallados, aprovechando las características que nos ofrecen las aceleradoras gráficas.

Planificando el movimiento

Los movimientos de los distintos personajes

En el número anterior ya establecimos la sólida base sobre la que programaremos nuestra aventura gráfica. Controlamos las acciones del menú, acciones entre objetos y los diálogos y monólogos. Todas estos sucesos se representarán a lo largo de los escenarios de nuestro juego.

Antes de comenzar a trabajar con los escenarios debemos planificar adecuadamente cómo se moverán los personajes en ellos. De todos los personajes, es el protagonista el que más tipos de movimientos realizará. El resto de personajes disponen de unos movimientos mucho más limitados. Tal y como definimos nuestra aventura al principio, el protagonista debe ser capaz de subir por unas escaleras para recoger una pelota del tejado del cobertizo, sin embargo, nuestro compañero de habitación jamás lo hará. De la misma forma, nuestro protagonista es capaz de andar en las cuatro direcciones, agacharse para coger cosas, hablar, etc.

Una forma sencilla de programar los movimientos sin tener que complicar el código es hacerlos lo más estándar posible

Para conseguir el efecto de movimiento en una "secuencia" debemos usar distintos "fotogramas". Nuestra secuencia de movimiento será la animación mínima que repetiremos tantas veces como sea necesaria para conseguir el movimiento completo. Cada fotograma es una imagen independiente, que se irá sustituyendo de forma ordenada para completar la secuencia. Un claro ejemplo es el movimiento del personaje al andar. Si hacemos secuencias de seis fotogramas, podríamos hacer:

1. *Pie izquierdo en el suelo, pie derecho ligeramente adelantado.*
2. *Pie izquierdo en el suelo, pie derecho adelantado.*
3. *Pie izquierdo cerca del derecho, pie derecho en el suelo.*
4. *Pie izquierdo ligeramente adelantado, pie derecho en el suelo.*
5. *Pie izquierdo adelantado, pie derecho en el suelo.*
6. *Pie izquierdo en el suelo, pie derecho cerca del izquierdo.*

De esta forma podemos crear una secuencia de animación sustituyendo la foto 1 por la 2, la 2 por la 3, la 3 por la 4 y sucesivamente. Esta secuencia podemos repetirla tantas veces como precisemos. Pero no todos los movimientos requieren el mismo número de imágenes, podemos conseguir un buen efecto de mover la boca al hablar con sólo cinco, o puede ocurrir que necesitemos muchos más para un movimiento más complejo.

Una forma sencilla de programar los movimientos sin tener que complicar el código es hacerlos lo más estándar posible. Definir cada tipo de movimiento y que éste sea igual para todos los personajes, es decir, todos andan con secuencias de seis imágenes o todos hablan con secuencias de cinco imágenes. De esta manera, tendremos una imagen base para el movimiento (la primera de ellas) y una longitud, número de imágenes en la secuencia. También nos simplificaría mucho que esta imagen base se calculase a partir de la imagen actual del personaje. Supongamos que hemos realizado el siguiente fichero FPG:

001 Mirando al frente
002 Anda de frente 1
003 Anda de frente 2
004 Anda de frente 3
005 Anda de frente 4
006 Anda de frente 5
007 Anda de frente 6
008 Habla de frente 1
009 Habla de frente 2
010 Habla de frente 3
011 Habla de frente 4
012 Habla de frente 5
013 Mirando a la derecha
014 Anda hacia la derecha 1
015 Anda hacia la derecha 2
016 Anda hacia la derecha 3
017 Anda hacia la derecha 4
018 Anda hacia la derecha 5
019 Anda hacia la derecha 6
020 Habla hacia la derecha 1
021 Habla hacia la derecha 2
022 Habla hacia la derecha 3
023 Habla hacia la derecha 4
024 Habla hacia la derecha 5


Fig. 1. Determinados movimientos complejos requerirán bastantes fotogramas.


Fig. 2. Mientras más tipos de movimientos implantemos a nuestro personaje más real se verá.

Con el fichero anterior sabemos que, según el estado del personaje (de frente o mirando hacia la derecha), nos basta sumar uno para obtener la imagen base de la secuencia andar o sumar siete para alcanzar la imagen base de la secuencia hablar. Construyendo el fichero FPG en este orden podemos conseguir que nuestro personaje ande y hable en las cuatro direcciones con el mismo código. Bastará con que, por ejemplo, al hablar se sume el "offset" correspondiente y el personaje hablará en la dirección en la que se encontraba mirando.

Este sistema implica un detalle muy importante a tener en cuenta: la numeración debe ser respetada con máximo rigor. Puede que un determinado personaje no ande ni hable de frente pero si lo haga a la derecha. No necesitamos poner las imágenes no utilizadas, pero las sí usadas deben tener la numeración, cómo si existiesen. Algunos personajes realizarán movimientos especiales, como por ejemplo nuestro compañero, que llorará. Estos movimientos especiales podemos situarlos al final del FPG.

Lo más cómodo para nosotros es hacer un fichero FPG por cada personaje o grupo de personajes. Esto nos permitirá cargar en memoria sólo aquello que estemos usando y descargar de memoria los gráficos que ya no nos hagan falta. Por otro lado, es mucho más cómodo para nosotros, ya que, por ejemplo, para el personaje protagonista tendríamos seis imágenes por cada uno de los cuatro movimientos, cuatro posturas base (el personaje mirando para cada lado) y cinco imágenes para hablar en cada dirección. En total veintiocho imágenes, sin contar otras secuencias como abrir puertas, coger, empujar o usar objetos. Incluso, para determinados movimientos muy especiales, convendrían que estuviesen en un fichero distinto. Ante todo debemos buscar dos cosas: hacerlo simple para nosotros y no ocupar memoria con lo que no tenemos previsto usar.

Gracias al orden que hemos establecido en nuestro fichero, una vez que el proceso del personaje sabe el número de la imagen a usar, será capaz de calcular las demás mediante el uso de un "offset". En nuestro ejemplo de fichero la imagen básica del personaje sería la número uno. A partir de ahí podemos hacer:

IMAGEN	OFFSET	BASE	POSICIÓN
Base del personaje	-	-	1
Mirar al frente	0	base	0
Mirar a la derecha	12	base	12
Andar	1	actual	1 (si miraba al frente)
			13 (si miraba a la derecha)
Hablar	7	actual	7 (si miraba al frente)
			20 (si miraba a la derecha)



Fig. 3. Para luchar a espada puede que nos basten cuatro o cinco fotogramas.

Ejemplo práctico

Tomando como base el programa del artículo anterior, nos bastan unas pequeñas modificaciones para hacer que los personajes gesticulen al hablar. Creamos las cinco imágenes correspondientes al movimiento "hablar" para cada uno de los personajes y modificamos ligeramente el código. En primer lugar, definimos las constantes que nos permitirán saber dónde empieza la secuencia hablar y el número de imágenes que la forman. La siguiente modificación es en el proceso *habla_ejecuta*. Al comenzar a hablar, recogemos la imagen actual correspondiente al personaje. En cada iteración (frame del proceso) calculamos la siguiente imagen de la secuencia de "forma circular". Una vez que dejemos de hablar no debemos olvidar asignar la imagen original al proceso para que el personaje quede como estaba inicialmente.

- *spr_hablar_inicio:* Indica el offset para la primera imagen (base) de la secuencia, se calcula a partir de la imagen base del personaje.
- *spr_hablar_total:* Número total de imágenes (fotogramas) en la secuencia. En el momento de visualizarlas será la número 0 (cero) la primera y *spr_hablar_total-1* la última.
- *sprite_org:* Almacena la imagen que posee el proceso en el instante en el que se empieza a hablar. Nos sirve para restablecerla una vez que finalice la acción y para usarlo como base en el cálculo de la imagen a representar.
- *sprite_n:* Es el número de la imagen de la secuencia a mostrar. Es el offset a sumar a la imagen base de la secuencia.

El siguiente número de imagen de la secuencia lo calculamos mediante la forma:

(sprite_n+1) mod spr_hablar_total. De esta forma conseguimos, si *spr_hablar_total=5* e inicialmente *sprite_n=0*, los valores: *1,2,3,4,0,1...* Su correspondencia en el fichero FPG será: *sprite_org + spr_hablar_inicio + sprite_n.*

Puesto que a cada uno de los personajes le corresponde una imagen diferente, *sprite_org* valdrá de forma diferente para cada uno de ellos. El uso de offset nos permite calcular la imagen a mostrar a partir de una «base», con indiferencia de su posición real en el fichero. Si las imágenes de los personajes estuviesen en ficheros FPG diferentes en vez de compartir el mismo, no necesitamos modificar el código de la función hablar. Bastaría con indicarle al proceso el fichero a usar y el número de la imagen inicial.

Cada fotograma es una imagen independiente, que se irá sustituyendo de forma ordenada para completar la secuencia

En el próximo artículo comenzaremos a trabajar con escenarios, situando personajes y objetos en él.

Miguel Adolfo Barroso
Barroso@almianos.net



Fig. 4. Los efectos especiales también pueden conseguirse a base de fotogramas.

Curso de juegos de rol

Isometría

Seguimos nuestro cursillo intensivo para programar juegos del género de rol y, continuando con la temática del artículo de la anterior edición de Divmanía, esta vez analizaremos la perspectiva isométrica.

Podemos distinguir 2 tipos de motores isométricos:

- **Tileados:** estos motores dibujan el mapa mediante tiles. El método de tileo isométrico se encuentra en el cuadro Tileo Isométrico.
- **No tileados:** en vez de usar tiles utilizan mapas ya dibujados en perspectiva isométrica.

La isometría pretende conseguir un efecto 3D mediante imágenes que son 2D, para conseguir esto se usan mapas isométricos (ver cuadro Mapas isométricos), que posteriormente serán tileados para obtener los resultados adecuados.

Para crear un mapa mediante los tiles usaremos una estructura que nos permita indicar qué tiles habrá en cada casilla (Consultar cuadro Estructura del Mapa).

Uno de los problemas que nos encontraremos es que desconocemos a qué tile pertenecen unas coordenadas x,y. Para resolver dicho problema podéis consultar el cuadro "Cómo averiguar en que tile estamos".



Imagen de *X-Com: Ufo Defense*, donde podemos encontrar un engine basado en tiles.

Una vez visto esto, la comprensión del siguiente ejemplo será mucho más sencilla.

```
program isometria;

  Global  Global
    tiles;
    struct isotiler
        string version=
"MaQIsOTiLeR 1.0";
// Version del tileador
        string nombre_mapa=
"Divmania 8";
// Nombre del mapa
        string fpg="iso.fpg";
// Fpg con gr ficos
        int NtileX;
// Longitud del Array
        int NtileY;
// Longitud del Array
        int tileX;
// Ancho de cada tile
        int tileY;
// Alto de cada tile
        int StartXTile;
// Tile desde el cual se empieza tilear
        int StartYTile;
// Tile desde el cual se empieza tilear
        int map;
// Mapa que usaremos para ir pintando antes de ponerlo en pantalla
        int map_aux;
// Mapa que usaremos para borrar el mapa map
        int CoordX;
// Cordenada X en al que se empieza a pintar
        int CoordY;
// Coordenada Y en la que se empieza a pintar
        int JugX
// Coordenada X del jugador
        int JugY;
```



Esto sería un tile isométrico. El rombo en sí tiene unas dimensiones de 64x31.

```
// Coordenada Y del jugador
        int aTX;
        int aTY;
        struct mapa[20,11]
// Array bidimensional con la informacion de cada punto dle mapa
            int L1tile;
            int L2tile;
            int L3tile[2];
            byte andable;
        end
    end

  Begin

    load("a.a",&isotiler);
    tiles=load_fpg(isotiler.fpg);
    isotiler.map=new_map
(320,240,160,120,0);
    isotiler.map_aux=new_map
(320,240,160,120,1);
    set_mode(m320x240);
    isotiler.CoordX+=isotiler.tilex/2;

    loop
      //

get_tile(isotiler.jugx,isotiler.jugY);
        write_int(0,0,10,0,
&isotiler.atx);

write_int(0,0,0,0,&isotiler.aty);

      //

      map_put(0,isotiler.map,
isotiler.map_aux,160,120);
      tilea(0);
      tilea(1);
      jugador();
      tilea(2);
```

```
        put_screen(0,isotiler.map);
        frame;
        teclado();
    end

    unload_fpg(tiles);
End

//////////////////////////////////////
//////////////////////////////////////
//////////////////////////////////////
//////////////////////////////////////

function get_tile(x,y)

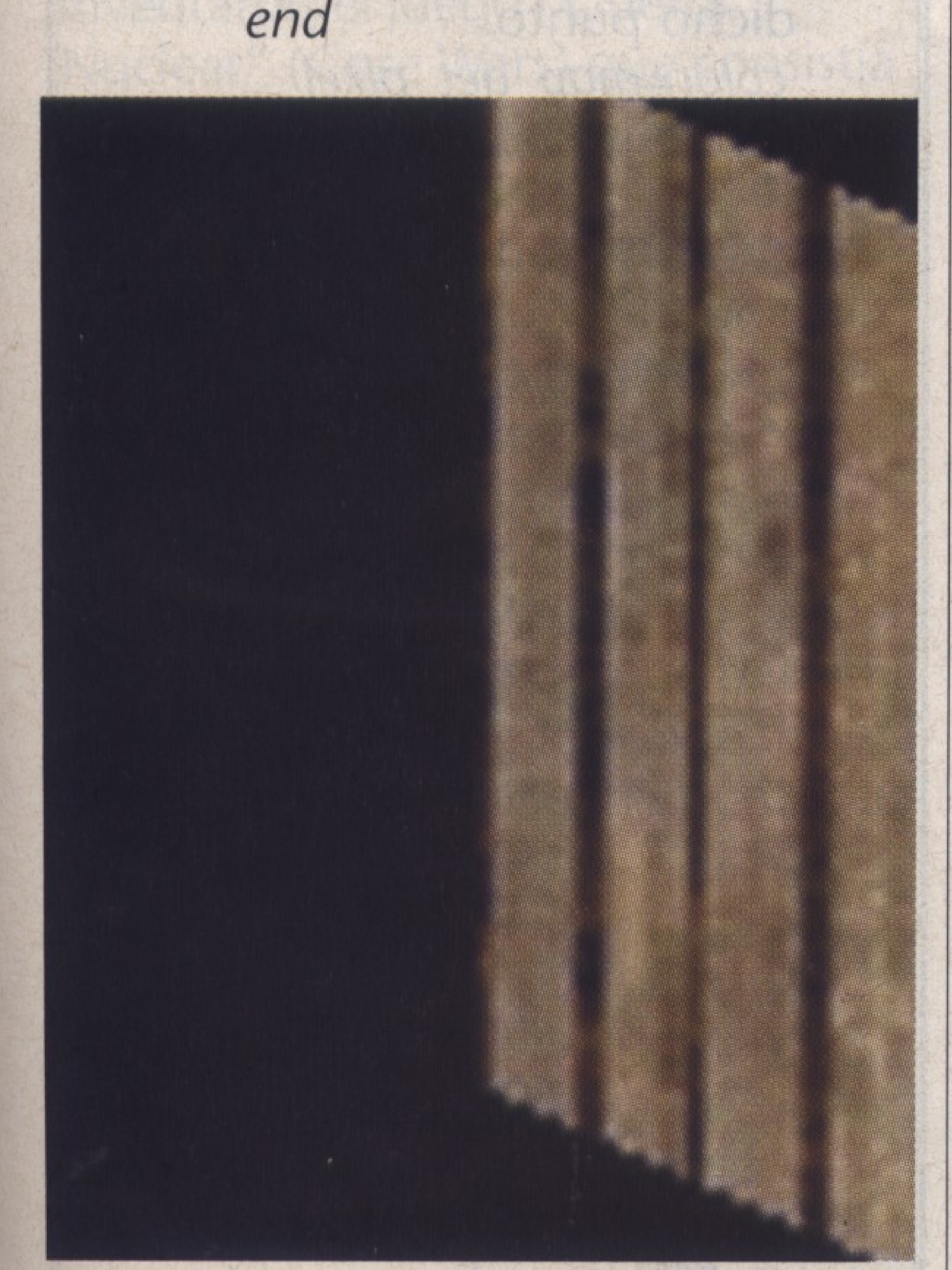
    Private
        regionX;
        regionY;
        mapX;
        mapY;
        color;
    Begin
        y-=isotiler.CoordY;
        x-=isotiler.CoordX-isotiler.tileX/2;
        regionX=x/isotiler.tileX;
        regionY=(y/isotiler.tileY)*2;
        mapX=x mod isotiler.tileX;
        mapY=y mod isotiler.tileY;
        color=map_get_pixel
(0,8,mapx,mapy);
        switch(color)

            case 195:
                regionX--;
                regionY--;
            end

            case 204:
                regionY--;
            end

            case 229:
                regionX--;
                regionY++;
            end

            case 74:
                regionY++;
            end
```



Se deja espacio por arriba para conseguir mapas como éste.

```
        end

isotiler.atx=regionX+isotiler.StartYTile
;
        isotiler.aty=regionY+isotiler.
StartXTile;
    End

//////////////////////////////////////
//////////////////////////////////////
//////////////////////////////////////
//////////////////////////////////////

function teclado()


begin

            if(key(_right))
                get_tile(isotiler.JugX+25,
isotiler.JugY);
                if(isotiler.mapa
[isotiler.aty,isotiler.atx].andable==0)
                    isotiler.Jugx+=5;
                end
            end

            if(key(_left))
                get_tile(isotiler.JugX-
25,isotiler.JugY);
                if(isotiler.mapa
[isotiler.aty,isotiler.atx].andable==0)
                    isotiler.JugX-=5;
                end
            end

            if(key(_up))
                get_tile(isotiler.JugX,
isotiler.JugY-25);
                if(isotiler.mapa
[isotiler.aty,isotiler.atx].andable==0)
                    isotiler.JugY-=5;
                end
            end

            if(key(_down))
                get_tile(isotiler.JugX,
isotiler.JugY+25);
                if(isotiler.mapa
[isotiler.aty,isotiler.atx].andable==0)
                    isotiler.JugY+=5;
                end
            end

            if(key(_esc))
                exit(0,0);
            end
end

//////////////////////////////////////
//////////////////////////////////////
//////////////////////////////////////
//////////////////////////////////////


Function jugador()

    Begin
        map_put(0,isotiler.map,10,
isotiler.JugX,isotiler.JugY);
    End
```



Captura de *Diablo 2*, donde se puede apreciar un engine isométrico no tileado.

```
//////////////////////////////////////
//////////////////////////////////////
//////////////////////////////////////
//////////////////////////////////////


Function tilea(int m)

    Private
        int i,j;
        byte linea;
    Begin

        y=isotiler.CoordY;
        x=isotiler.CoordX;

        if(m==0)
            for(i=isotiler.startXTile;
i<isotiler.startXTile+14 && i<isotiler.
NtileX;i++)
                for(j=isotiler.startYTile;
j<isotiler.startYTile+7 &&
j<isotiler.NtileY;j++)
```

Mapas isométricos

Dichos mapas no son rectángulos, sino rombos (un rombo es aquella figura que tiene sus lados iguales y sus ángulos iguales dos a dos).
Tileo Isométrico: el orden de los tiles debe ser de izquierda a derecha y de arriba abajo. El dibujo de los tiles se hace línea a línea, y para ello utilizaremos un doble bucle.

```
end
y+=tileheight/2
    if(linea==0)
        linea=1;
        x=tilewidth/2;
    else
        x=0
        linea=0;
    end
end
```

Este trozo de código lo que hace es pintar todos los tiles de una línea, bajar la mitad de la altura de un tile, y según sea la línea par o impar empieza en x=0 o en la mitad del ancho del tile.

Estructura del mapa

La estructura del mapa debería ser algo así:

```
Struct mapa[NtileX,NtileY]
        Int L1Tile;
        Int L2tile;
        Int L3Tile[2];
    Int andable;
End
```

Cada estructura del array bidimensional contiene la información de qué gráficos contendrá. Estos están divididos en 3 capas:
Capa 1: En esta capa se almacena el código del grafico del suelo que habrá en esa posición.
Capa 2: En esta capa se almacena el código del grafico que modificará el suelo.
Capa 3: La capa 3 puede tener hasta 3 gráficos. Estos gráficos pueden ser paredes, techo, etc.
Por lo general no se suelen usar más gráficos, pero si es necesario no hay ningún problema en añadir más capas o más gráficos a las capas ya existentes.
Solo usamos 3 gráficos distintos en la capa 3 dado que las únicas combinaciones que queremos son las de las fotos 6, 7 y ocho.
Cada tile tendrá también la variable andable que indicará si se puede andar por un tile o no.

```
            if(isotiler.mapa[i,j].
L1tile!=0)
                map_put(0,
isotiler.map,isotiler.mapa[i,j].L1tile,x,y);
            end

if(isotiler.mapa[i,j].L2tile!=0)
                map_put(0,
isotiler.map,isotiler.mapa[i,j].L2tile,x,y);
            end
            x+=isotiler.tileX;
        end
        if(linea==0)
            x=isotiler.CoordX+
isotiler.tilex/2;
            linea=1;
        else
            x=isotiler.CoordX;
            linea=0;
        end
        y+=isotiler.tileY/2;
    end

    else
        for(i=isotiler.startXTile;
i<isotiler.startXTile+14 &&
i<isotiler.NtileX;i++)
for(j=isotiler.startYTile;j<isotiler.startY
Tile+7 && j<isotiler.NtileY;j++)

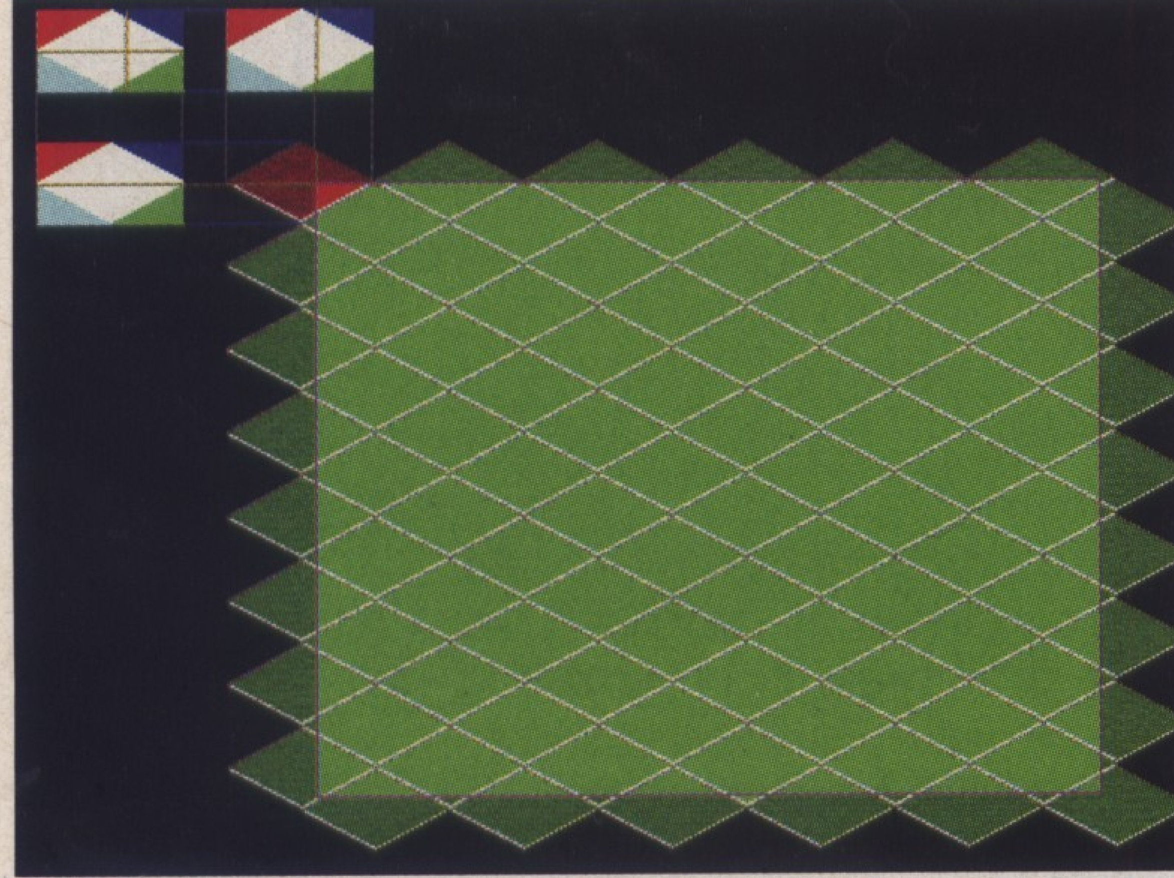
            if((y+10)>isotiler.JugY
&& m==1)break;end
            if((y+10)<isotiler.JugY
&& m==2)break;end

if(isotiler.mapa[i,j].L3tile[0]!=0)
                map_put
(0,isotiler.map,isotiler.mapa[i,j].L3tile
[0],x,y);
            end

if(isotiler.mapa[i,j].L3tile[1]!=0)
                map_put
(0,isotiler.map,isotiler.mapa[i,j].L3tile
[1],x,y);
            end

if(isotiler.mapa[i,j].L3tile[2]!=0)
                map_put
(0,isotiler.map,isotiler.mapa[i,j].L3tile
[2],x,y);
            end

            x+=isotiler.tileX;
        end
        if(linea==0)
            x=isotiler.CoordX+
isotiler.tilex/2;
            linea=1;
        else
            x=isotiler.CoordX;
            linea=0;
        end
        y+=isotiler.tileY/2;
    end
    end
End
```



La parte de color verde claro es la que se verá en la pantalla (foto 1).

El código no debería resultar muy complicado de entender, y quizás el único problema en su entendimiento sea la resolución del problema que uno se puede encontrar en el manejo de objetos que cubran al jugador. Para solucionar el problema se ha separado el tileo en tres etapas:

1ª Etapa: se tilean únicamente los tiles que siempre estarán por debajo del jugador.

2ª Etapa: se pintan todos aquellos objetos que están por debajo del jugador. Se pinta el jugador.

3ª Etapa: se dibujan todos aquellos objetos que taparán al jugador.

Para distinguir que tiles taparán al jugador, lo hacemos mediante la coordenada *y* (que equivaldría a la *z*), es decir, todos aquellos objetos que tengan una y inferior a la del jugador irán por debajo del jugador, y todos aquellos que tengan una y superior irán por encima.

Ramón de España (MaQNaZiLLeR)
MaQNaZiLLeR@pagina.de

¿Cómo averiguar en que tile estamos?

Para hallar el tile en el que nos encontramos utilizaremos el mapa de la foto 1. El proceso para hallar el tile consiste en:

a) Dividir la pantalla en regiones rectangulares del tamaño de un tile.
regionX=x/isotiler.tileX;
*regionY=(y/isotiler.tileY)*2;*

b) Hallar las coordenadas x, y en dicha región.
mapX=x mod isotiler.tileX;
mapY=y mod isotiler.tileY;

c) Hallar el color del mapa en dicho punto.
color=map_get_pixel (0,8,mapx,mapy);

Según el color cambiamos la región.

```
switch(color)
    case 195:
        regionX--;
        regionY--;
    end
    case 204:
        regionY--;
    end
    case 229:
        regionX--;
        regionY++;
    end
    case 74:
        regionY++;
    end
end
```

Virología

¿Nuestros ordenadores en peligro?

El fenómeno virus, esta cada vez más en auge debido a los recientes sistemas operativos de 32 bits y sobre todo al fenómeno de Internet. Cada vez se hace más necesario la protección de nuestros ordenadores con antivirus y la continua actualización de éstos para evitar todo tipo de virus... como el ya famoso "I love you".

Mucha gente no conoce realmente qué es un virus, cómo funciona, o qué tipos hay. En este reportaje pasaremos a explicar la historia de estos engendros y todas sus partes.

La historia

La primera definición de virus fue dada por John Von Neumann en el año 1949, un experto en teoría de ordenadores, expuso su teoría sobre programas que eran capaces de multiplicarse en un artículo llamado "Teoría y organización de un autómata complicado". Por aquellos tiempos, nadie supuso la repercusión que provocaría dicho artículo.

Diez años mas tarde, en 1959, los laboratorios llamados AT&T Bell inventaron el juego "Guerra Nuclear" (Core Wars) que consistía en una batalla entre los códigos de dos programadores, cada uno de estos programadores debía desarrollar un programa cuyo fin era el de acaparar la máxima memoria posible del contrincante mediante la reproducción de sí mismo.

En este juego, cada uno de los programas intentaba destruir al del oponente y tras un tiempo determinado ganaba el jugador que tuviera la mayor cantidad de memoria ocupada con su programa.

En 1983 este juego contaba con muchos adeptos y salió a la luz pública en un discurso de Ken Thompson en la entrega del premio Turing. Ese mismo año aparece la definición del termino virus tal y como se conoce hoy en día, Fred Cohen lo definió como "un programa que puede infectar otros programas modificándolos para incluir una versión de sí mismo". Diseño varios experimentos donde mostraba la factibilidad de estos engendros y demostró las limitaciones que se tenían en aquella época para defenderse de ellos, a la vez que la imposibilidad de diseñar un sistema de detección universal.



Es definitivamente en los años 1986-87 cuando se produce el fenómeno virus en PCs. Fue en el entorno universitario donde se detectaron los primeros casos de infecciones masivas. Los primeros protagonistas fueron el virus Brain, el Bethlehem y el conocidísimo Viernes 13.

Los virus y sus variantes

Los virus son indiscutiblemente los programas más dañinos que existen. Pero no debemos por ello olvidarnos de otros programas existentes que pueden igualmente causarnos destrozos en nuestros sistemas.

Así, por ejemplo, tenemos a los "gusanos" y "conejos", que son programas que comparten con los virus su principal característica, la de la su reproducción. Estos programas tienen como objetivo realizar múltiples copias de sí mismo, lo que terminará por desbordar y colapsar el sistema.

El gusano más famoso y conocido fue el de Robert Morris, que consiguió bloquear la red ARPAnet. No fue casualidad que su padre fuera uno de los implicados en el desarrollo de Unix y pionero en el "Core Wars".

Una de las peores capacidades de estos virus "gusanos" es que son capaces de propagarse muy rápidamente a través de las redes.

Otro tipo de variante de los virus son los "Caballo de Troya" o más conocidos como "Troyanos" este nombre se debe a que, al igual que en el acontecimiento mitológico, estos dañinos programas se presentan en forma de aplicación "normal", pero en su interior poseen un código destructivo.

Pero hay que decir que los Troyanos, a diferencia de los virus, no tienen la capacidad de replicación o reproducción. Se suelen presentar como utilidades comunes que todos utilizamos, por lo que podemos encontrarnos Troyanos que simulen compresores (ARJ, Winzip) o incluso antivirus (McAfee, Panda, etc). Uno de los troyanos que más impacto causo, por su repercusión en los medios de comunicación fue el conocido AIDS.

También existen las "Bombas Lógicas", que son programas que se ejecutan al producirse un hecho determinado, este hecho puede ser una fecha, combinaciones de teclas, etc., y si no se produce este hecho el programa permanece oculto sin ejercer ninguna otra acción. Esta técnica es, en ocasiones, utilizada fraudulentamente por programadores en sus aplicaciones. Consiguen de esta manera que en determinadas fechas el programa genere un error, de manera que el cliente se ve forzado a recurrir al programador para corregir el error y asegurándose de esta forma el mantenimiento.

No podemos, cómo no, olvidarnos de los virus, programas que utilizan características combinadas de los "Troyanos", "gusanos" y "bombas lógicas". De esta forma, se reproducen, se introducen en aplicaciones originales y pueden causar diferentes efectos cuando se producen determinadas condiciones.

Por último, los applets Java y ActiveX dados por lenguajes orientados a Internet, han permitido la potenciación y flexibilidad en la Red. Sin embargo, estas tecnologías abren un nuevo mundo para los creadores de virus. Aunque todavía no se ha producido un uso masivo de estas técnicas, pruebas que han sido realizadas demuestran la factibilidad del uso de estos lenguajes para realizar las funciones de los programas comentados anteriormente.

Actualmente hay circulando unos diez millones de virus, un antivirus es esencial para protegernos

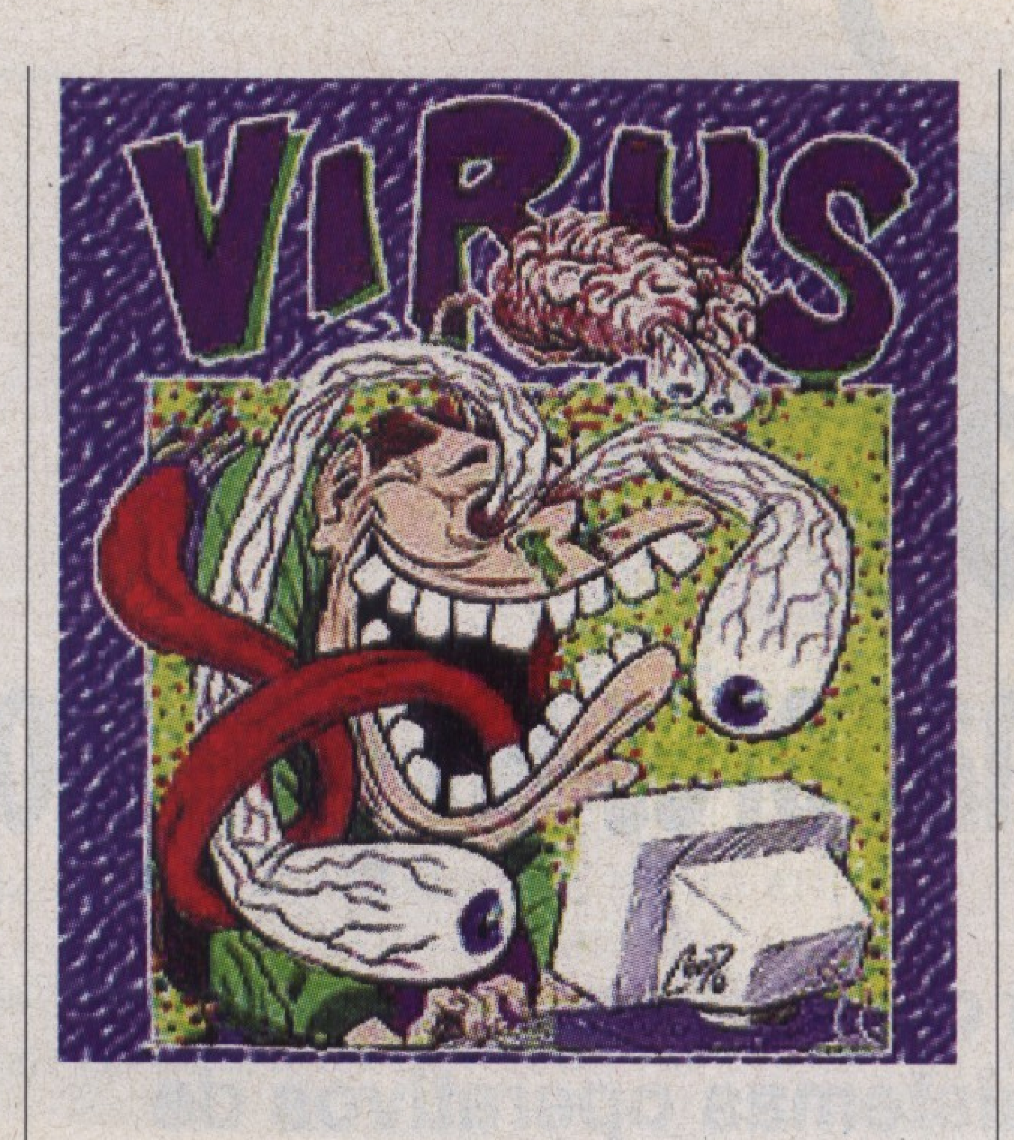
Actualmente la mayoría de las casas antivirus vienen dando a sus productos un enfoque orientado a la Red, para comenzar a detectar applets Java y controles ActiveX perjudiciales o maliciosos.

Funcionamiento de un virus

Hay que tener siempre en cuenta que un virus es simplemente un programa. Ya se han dado casos de personas que han recurrido a centros médicos con sus equipos por que decían que tenían un "virus".

Por lo tanto, debemos dejar a un lado las histerias y los miedos infundados y al mismo tiempo ser conscientes del daño real que pueden causarnos. Por ello, lo mejor es tener un buen conocimiento de qué son estos programas, cómo funcionan y las medidas que debemos tomar para prevenirlos y hacerles frente.

El nacimiento de cualquier virus viene dado por personas que evidentemente tienen que tener un alto grado de conocimientos de programación. Las motivaciones que llevan a estas personas a crear



estos engendros son muy variadas y no vamos a entrar en ese tema en este artículo, aunque hay rumores de que la mayoría de los virus actuales son creados por las propias empresas antivirus para asegurarse así la permanencia de su producto en el mercado.

Estos programas pueden desarrollarse en muchos lenguajes de programación distintos, aunque el más usado para este fin, sin duda, es el ensamblador por su potencia y eficacia.

El objetivo de todo virus es el de replicarse o reproducirse a sí mismo de forma que el usuario no lo note y dificultar al máximo su detección. Para poder replicarse necesita ser ejecutado en el ordenador, por lo que suele recurrir a los programas o ficheros ejecutables, uniéndose a ellos y modificándolos o si no situándose en los sectores de arranque y tablas de particiones de los discos.

Una vez que se ejecutan, ya sea por abrir un fichero o archivo infectado o por hacer una operación de un disco con el boot infectado, suelen quedar residentes en memoria a espera de poder infectar otros ficheros y discos. Los virus llamados residentes interceptan los vectores de interrupción, modificando la tabla que los contiene para que apunten a su código y los ejecuten. Los vectores son los encargados de prestar servicios del sistema; de esta manera, cuando una aplicación llame a uno de esos servicios, el control es cedido al virus.

Una vez que el virus tiene el control del sistema se dispone a la replicación, ya que una llamada al servicio de ejecución o copia de un fichero puede ser interceptada gracias a las modificaciones de los vectores de interrupción y proceder a la infección.

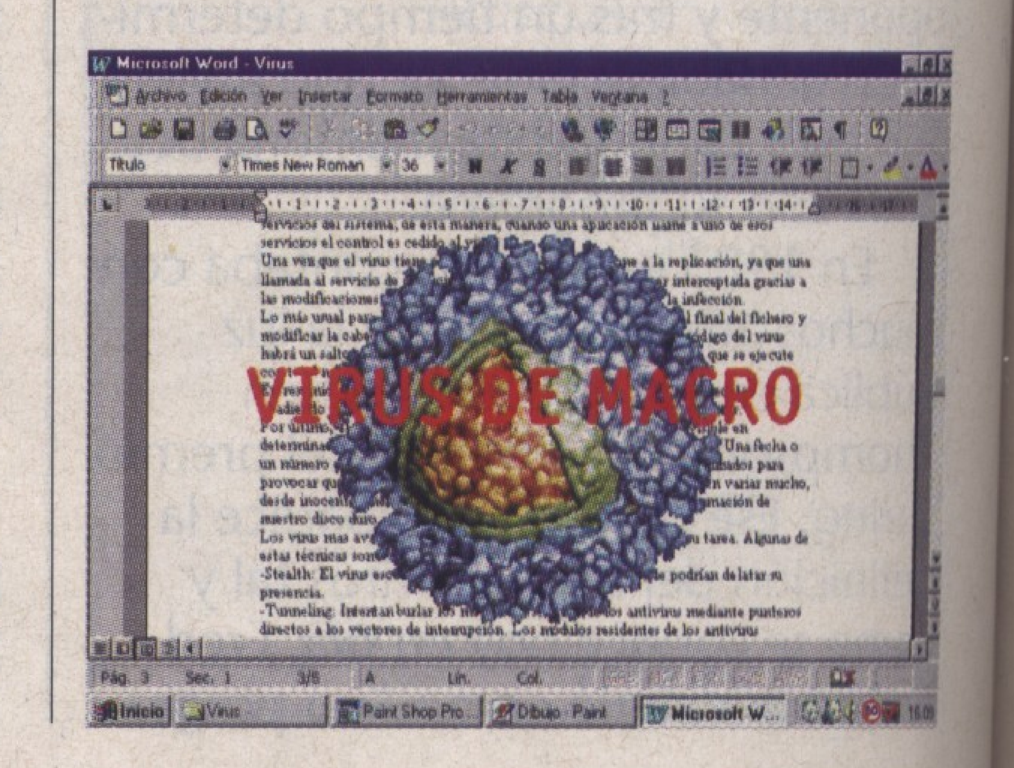
Lo más usual para hacer esto consiste en añadir al código vírico al final del fichero y modificar la cabecera de este para que apunte al virus. Al final del código del virus habrá un salto al comienzo del programa o fichero original de manera que se ejecute con total normalidad para que el usuario no sospeche.

En resumidas cuentas, lo que hace es modificar cada programa que se ejecuta añadiendo su código de virus a éste, pero sin impedir que se pueda ejecutar.

Por último, el virus suele contener algún tipo de efecto que se hará visible en determinadas circunstancias para que el usuario sepa que está infectado. Una fecha o un número concreto de infecciones suelen ser comúnmente los más utilizados para provocar que aparezca ese efecto. Los tipos de efectos utilizados pueden variar mucho, desde inocentes mensajes en pantalla, hasta la perdida total de la información de nuestro disco duro.

Los virus más avanzados utilizan técnicas para hacer más efectiva su tarea. Algunas de estas técnicas son:

- **Stealth:** el virus esconde los signos visibles de la infección que podrían delatar su presencia.
- **Tunneling:** intentan burlar los módulos residentes de los antivirus mediante punteros directos a los vectores de interrupción. Los módulos residentes de los antivirus funcionan de forma muy parecida a los virus, interceptando los servicios del sistema, pero lógicamente con un propósito totalmente diferente.
- **Autoencriptacion:** Permite al virus que se encripte de manera diferente cada vez que este infecte un fichero. De esta manera dificulta la labor de detección del antivirus. Normalmente son detectados por la presencia de la rutina de desencriptación ya que ésta no varía.
- **Poliformismo:** La contramedida de los virus para impedir ser detectados mediante la desencriptación es variar el método de encriptación de generación en generación. Es decir, que entre distintos ejemplares del mismo virus no existen coincidencias ni siquiera en la parte del virus que se encarga de la desencriptación; son los llamados virus polimórficos.





En esta guerra abierta, los creadores de virus en su afán de no ser detectados por los antivirus llegan a implementar técnicas específicas para burlarlos.

Tipos de virus

Se pueden diferenciar distintos tipos de virus dependiendo del lugar donde se alojen, la técnica de replicación o la plataforma en la que trabajen.

Los virus de Boot, utilizan el sector de arranque, que es el que contiene toda la información sobre el tipo de disco (sectores, caras, número de pistas, etc.) y un pequeño programa para verificar si se puede cargar el sistema operativo, estos virus utilizan este sector para ubicarse guardando el sector original en otra parte del disco. En muchas ocasiones el virus marca los sectores donde se guarda el boot original como defectuosos para impedir que sean borrados.

En el caso de los discos duros pueden utilizar también la tabla de particiones como ubicación. Suelen quedar residentes en memoria al hacer cualquier operación en un disco infectado a la espera de replicarse en otros.

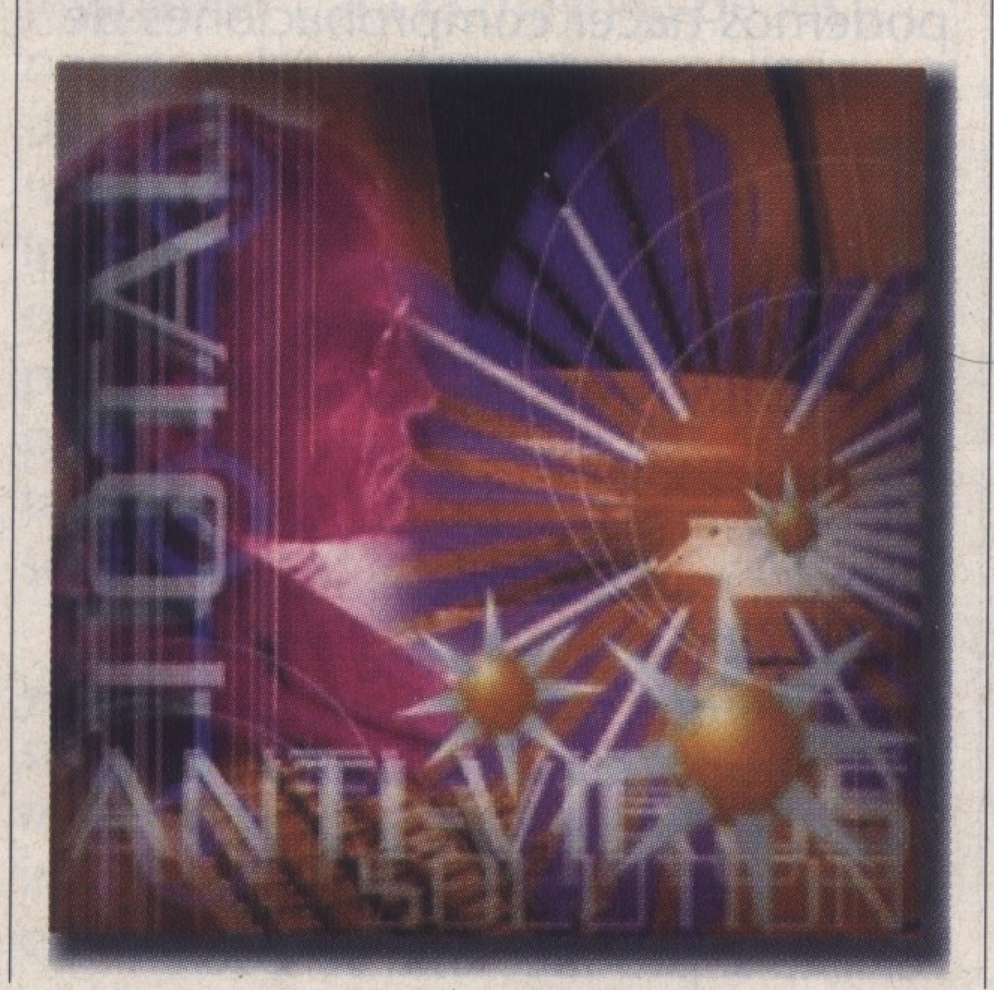
Los virus de fichero, como su propio nombre indica, infectan archivos y, tradicionalmente, los tipos ejecutables COM y EXE han sido los más afectados, aunque en estos momentos son los ficheros de documentos (DOC, XLS, SAM...) los que están en auge gracias a los virus de macro. Normalmente insertan el código del virus al principio o al final del archivo, manteniendo intacto el programa infectado. Cuando se ejecuta, el virus puede hacerse residente en memoria y luego devuelve el control al programa original para que continúe de modo normal y el usuario no sospeche. El Viernes 13 es un ejemplar representativo de este grupo.

Dentro de la categoría de virus de ficheros podemos encontrar más subdivisiones. Así, los virus de acción directa son aquellos que no quedan residentes en memoria y que se replican en el momento de ejecutarse un fichero infectado. Los virus de sobreescritura corrompen el fichero donde se ubican al sobreescribirlo.

Los virus de compañía aprovechan una característica del DOS, gracias a la cual si llamamos a un archivo para ejecutarlo sin indicar la extensión, el sistema operativo buscará en primer lugar el tipo COM. Este tipo de virus no modifica el programa original, sino que cuando encuentra un archivo tipo EXE crea otro de igual nombre conteniendo el virus con extensión COM. De manera que cuando tecleemos el nombre ejecutaremos en primer lugar el virus, y posteriormente éste pasará el control a la aplicación original.

Una familia de virus de reciente aparición y gran expansión son los virus de macro. Estos están programados usando el lenguaje de macros WordBasic, gracias al cual pueden infectar y replicarse a través de ficheros MS-Word (DOC). En la actualidad esta técnica se ha extendido a otras aplicaciones como Excel y a otros lenguajes de macros, como es el caso de los ficheros SAM del procesador de textos de Lotus. Se ha de destacar, en relación a este tipo de virus, que son multiplataformas en cuanto a sistemas operativos, ya que dependen únicamente de la aplicación. Hoy día son el tipo de virus que están teniendo un mayor auge debido a que son fáciles de programar y de distribuir a través de Internet, y aún no existe una concienciación del peligro que puede representar un simple documento de texto. Sin duda, el más extendido de este tipo de virus fue el Concept, gracias al descuido de Microsoft que lo incorporó accidentalmente en un CD, el cual se distribuyó por millares a mediados del año 1995.

Sin duda alguna, el ensamblador es el lenguaje por excelencia en la creación de virus. Ningún otro lenguaje puede ejercer tal control sobre el sistema, ni permite tan alto grado de optimización. En realidad, es prácticamente factible programarlos en cualquier lenguaje, desde C hasta Basic. Los virus BAT vienen a





reafirmar este hecho, ya que, empleando ordenes DOS en archivos de proceso por lotes, consiguen replicarse y realizar efectos dañinos como cualquier otro tipo de virus.

Hay que tener en cuenta que existe una gran mayoría de temores infundados por los virus, así por ejemplo los temidos virus de correo electrónico no existen, son simples rumores que se propagan, para que un virus te infecte el ordenador ha de ser autoejecutable el archivo en el que encuentre, y los mensajes de correo electrónico no lo son, así también sé ha llegado a decir que existen virus que pueden estropear algunos componentes de hardware de nuestro ordenador cosa que por el momento es imposible.

Por último, no todos los virus son malignos, aunque siempre se asocie el termino con destrucción. Podemos señalar que hay virus benignos y que todo depende de la utilidad que se le dé a esta técnica. Con el termino "virus benigno" no nos referimos a aquellos que tienen como payload (efecto del virus cuando se activa) alguna pantalla graciosa y no destruyen datos. Nos referimos a que una buena utilización de las técnicas que emplean los virus pueden, aunque desgraciadamente no es lo habitual, reportarnos beneficios y ser sumamente útiles.

> No todos los virus dañan nuestro ordenador, algunos simplemente son pantallas graciosas

Se puede utilizar estos virus para parchear sistemas a través de redes. El programa se infecta de ordenador a ordenador modificando parte de un programa que causaba fallos en el sistema, y una vez solucionado el error, este virus se autodestruye.

Se espera que en este año la cifra de virus en circulación pase a ser de diez millones de virus, por esta razón debemos tener siempre protegido nuestro ordenador con un buen antivirus actualizado y siempre tener cuidado de todos los archivos o ficheros que nos envíen por Internet, tanto ejecutables como documentos de texto.

Daniel García Alonso (PORTOX)
Dagal@Alehop.com

Realismo Vs. Jugabilidad

La esencia de los videojuegos

Quizás una de las preguntas más antiguas que se han podido realizar los programadores de videojuegos se basa en el tema del realismo y la jugabilidad: ¿cuál de los dos es más importante? Aún más: ¿el uso de uno excluye la otra? Quizás no exista una solución a este problema, pero vamos a verlo con más detalle.



El balance debe ser tratado con cuidado combinando realismo y jugabilidad.

Para empezar a tratar el tema, lo más importante sería conocer el significado de ambos términos, lo cual nos lleva a un tema muy complejo en el que los diseñadores no acaban de estar de acuerdo. Del mismo modo, los jugadores entienden estos dos conceptos de una manera bien distinta por lo tanto empecemos viendo lo que significan para los diseñadores de videojuegos:

Debemos encontrar la justa medida entre realismo y jubabilidad, así nuestros juegos no decepcionarán a nadie

Para un diseñador, un juego "realista" es aquel en el que se cumplen las siguientes condiciones o premisas:

Primeramente, se puede llamar a un juego "realista" porqué intenta hacer una analogía lo más cercana a la realidad. Por ejemplo en un juego de estrategia donde aparezca una unidad de infantería que pueda caminar un equivalente a 30 kilometros diarios por un buen camino sería realista. En un juego podemos hacer comprobaciones de su "índice de realidad" a partir de chequeos. Mientras más chequeos pase, cumpliendo leyes de la naturaleza, más realista será el juego. El inconveniente de los chequeos es que no podemos comprobar si son realistas aquellos aspectos que no pueden ser cuantificados, como el "daño": nadie sabe cuánto daño se puede ocasionar a una persona antes de que muera. Para ello deberemos utilizar la intuición en este tipo de chequeos.

Un juego puede ser llamado realista si produce una sensación global de realidad a pesar de que algunas de sus pautas no lo sean. Por ejemplo, se puede dar el caso de un juego de estrategia en el que aparezcan elementos poco precisos, pero que den una sensación de realismo al jugador. Algunos ejemplos son aquellos juegos que intentan seguir fielmente o mostrar aspectos reales de períodos de la historia o culturas como podrían ser la Segunda Guerra Mundial o el Antiguo Egipto. Si el juego intenta mostrar como eran fielmente esos hechos será un juego realista, a pesar de que algunos de sus subsistemas no lo sean en absoluto.

Finalmente, un diseñador de juegos puede llamar a un juego realista si da una sensación global que concuerde con algún tipo de ficción. Por ejemplo, los juegos ambientados en los mundos de "Dungeons & Dragons" son realistas conforme a las leyes que ambientan ese mundo de ciencia-ficción o fantasía.

Jugabilidad

Respecto a la jugabilidad, un diseñador dirá que un juego es "jugable" si cumple estas premisas:

Si el jugador puede aprender el funcionamiento del juego de una forma rápida y precisa. Obviamente, mientras más simples sean las reglas, más fácil serán de aprender. Pero debemos tener en cuenta también que unas reglas bien explicadas y un sistema de juego intuitivo pueden dar un mejor resultado. La inclusión de tutoriales o de períodos de aprendizaje en el juego pueden hacer que un título sea mucho más jugable.

En segundo lugar, algunos juegos de reglas complejas son jugables porqué una vez aprendidas las reglas es muy difícil que no entendamos alguno de los sistemas. Pongamos por ejemplo el juego *Dune 2*, quizás uno de los grandes predecesores de los juegos de estrategia en tiempo real: al princi-

pio podía parecer complicado, pero una vez aprendido el sistema de juego era muy fácil adaptarlo a los sistemas avanzados e incluso a juegos del mismo estilo como resultaron ser *Warcraft* o *Command & Conquer*. Muchas veces el funcionamiento de un juego se transmite de forma oral entre un grupo de jugadores, y si este sistema es fácil y se hace comprensible puede hacer que un juego de reglas complicadas resulte jugable.

Finalmente, un diseñador puede considerar jugable, de una forma mucho más técnica, si los diversos subsistemas del juego se mezclan correctamente y se muestran de una forma amigable, de fácil uso. Si el jugador se encuentra con que las diferentes piezas de un juego son difíciles de comprender y al mismo tiempo no acaban de encajar en el sistema global, solamente conseguiremos que el resultado sea algo "injugable", que el usuario rápidamente apartará de su mente.

El sufrido jugador

A continuación consideraremos las definiciones desde el punto de vista de los jugadores, después de todo para ellos es el juego que estamos diseñando.

Un jugador considerará que un juego es realista normalmente cuando:

A pesar de lo que el diseñador pueda pensar, un jugador que después de leer las instrucciones del juego crea que son complicadas y haya quedado impresionado por ellas seguramente pensará que el juego se aproxima a la realidad. Un diseñador podrá observar que muchos de los juegos realistas son complicados, pero no todos los juegos complicados son realistas y que, por ello, algunos jugadores pueden sentirme intimidados por grandes cantidades de reglas difíciles de comprender.



El mundo del rol tiene sus propias reglas reconocidas por todos los aficionados.

En segundo lugar, un jugador creerá que el juego es "realista" cuando, después de haber jugado con él, piense que no hay nada fuera de lugar o de funcionamiento estúpido. El jugador no se dedicará a realizar los chequeos de realidad que el diseñador ha hecho durante el proceso de creación, por lo tanto la simple concordancia de los elementos del juego pueden hacer que el jugador crea que es realista a pesar que éste no lo sea. Si durante el uso del producto el jugador observa elementos poco realistas o incluso discordantes con la realidad, tenderá a observar el juego de una manera mucho menos realista, a pesar que se trate de pequeños detalles.

Es muy importante tener en cuenta que el jugador, a la hora de juzgar el producto, lo hará basándose en su propia experiencia: él puede tener concepciones diferentes de la realidad que las del diseñador o incluso puede ser que domine algunos temas mucho más que los diseñadores de un juego y de esta manera encuentre defectos o elementos poco reales. Algunos jugadores versados en los RPGs están acostumbrados a que los personajes tengan una resistencia sobrehumana, esto puede ocasionar que si, por ejemplo, su personaje en el juego esté a punto de morir simplemente por un golpe bien colocado, sientan que sea poco realista a pesar de que cumpla las leyes de la naturaleza. Este jugador no tiene experiencia en una batalla real y simplemente tiene la concepción que le han mostrado las películas y los juegos.

Jugando sin parar

Si hablamos de jugabilidad, el jugador se sentirá satisfecho en este aspecto cuando haya disfrutado realmente del juego. En este caso "disfrutar" es sinónimo de "jugabilidad".

Por ejemplo, la mayoría de los juegos de rol son considerados injugables para la mayoría de los neófitos: las reglas son complicadas, se deben tener en cuenta muchos aspectos y se deben improvisar muchos de ellos. En cambio los jugadores que participaban en los juegos de rol se lo pasan realmente muy bien, por lo tanto los juegos de rol para ellos serán "jugables".

Como podemos ver "realismo" y "jugabilidad" son términos muy subjetivos que pueden tener diferentes significados para la gente. Es importante que un diseñador sea crítico cuando esté realizando su



El realismo histórico predomina junto a la jugabilidad.

trabajo e intente estar abierto al mayor número posible de significados y de mentalidades. Para ello es importante hacer un estudio simple sobre las interpretaciones a primera vista que un jugador pueda dar del aspecto global del juego.

Realidad y Jugabilidad: ajustando la balanza

Tras la primera exposición del significado de ambos términos, debe haber quedado claro qué es cada uno de ellos; es más, nos debe haber quedado la idea de que ambos son importantes. A pesar de todo, es muy difícil conseguir que un producto tenga grandes cantidades de ambos elementos.

> Muchos de los componentes "reales" son bastante subjetivos, lo mejor es usar el sentido común

En general, se acepta de alguna manera que en un juego extremadamente realista la jugabilidad será mucho menor y que al contrario en un producto de alta jugabilidad encontraremos algo menos realista. A primera vista esto es normalmente verdad, a pesar que no es ninguna ley que el realismo y la jugabilidad sean excluyentes en su totalidad y siempre se hallen en conflicto. El diseñador es el encargado de que el usuario final encuentre un equilibrio entre ambos para que el mercado consumidor sea mucho más grande. Normalmente el diseñador adoptará la proporción justa de cada elemento dependiendo del destinatario final del producto: para un niño pequeño no diseñaremos una enciclopedia precisa y compacta con todos los elementos al pie de la letra, sino que diseñaremos algo que sea agradable al uso y que al mismo tiempo sirva para que pueda aprender cosas. Del mismo modo, haremos lo contrario con una persona adulta. De esta manera la jugabilidad predomina en mercados juveniles y el realismo



La estrategia en tiempo real suele dar más sensación de realismo que la estrategia por turnos.

abunda en mercados adultos o en ámbitos de coleccionismo como pueden ser los apasionados por los "wargames".

Pasando a la práctica

Ahora que hemos escogido nuestra proporción de jugabilidad y realismo, ¿cómo podemos conseguir plasmarla en nuestro producto final? Hay muchos sistemas para conseguirlo pero todos dependen de las decisiones de diseño a la hora de crear el juego. Algunos ejemplos que podemos encontrar son los siguientes:

Tablas de efectos específicos. Un juego puede ignorar por ejemplo el tiempo, o puede tener por ejemplo un par de efectos climáticos como lluvia y nieve. O incluso un diseñador puede añadir un complejo sistema meteorológico que incluya docenas de situaciones diferentes que dependan de modificadores según los días que van pasando o la estación en que se encuentre el personaje.

Un juego excesivamente real puede quedar demasiado complicado para un usuario no avanzado

Localizaciones de los efectos. Muchos juegos incluyen puntos débiles en los enemigos e incluso hacen que algunos efectos sean más o menos potentes dependiendo de donde o contra que los utilicemos. El juego de lucha *Bushido Blade*, es quizás uno de los más realistas en lo que se refiere a lucha, donde una estocada bien colocada puede acabar con el adversario de un solo golpe.

En los wargames o simuladores, puede haber reglas complejas como un número limitado de munición o algunas limitaciones de movimiento. Algunas reglas más complicadas pueden incluir un campo de visión limitado, lo cual es realista, o aún más efectos sencillos pero efectivos como la famosa "niebla de batalla" que se utiliza en muchos juegos como *Warcraft*.

Incluso algunos juegos pueden hacer que el paso del tiempo sea realista o no. Con ello nos encontramos los sistemas de juegos por turnos o de tiempo real. Quizás en los juegos de estrategia o en los RPGs encontremos los ejemplos más claros. Juegos como *Civilization* incluyen un sistema por turnos a pesar que el tiempo transcurra progresivamente, otros títulos como *Simcity* utilizan un sistema de juego en tiempo real, a pesar de que nosotros podemos graduar su paso.

Consiguiendo Realismo

Muchas veces es difícil saber si un efecto es realista o no: saber si el golpe de una espada está programado para que cause daño de una forma realista es difícil de calcular, sobre todo cuando no podemos experimentar muchos de estos hechos. La mejor manera de solucionar este problema es mediante las leyes de causa-efecto que tenemos todos en mente. La mayoría de la gente no espera que una persona aguante una infinidad de golpes de espada antes de morir, por lo tanto con un número bajo de golpes conseguiremos efectos realistas debido a que la gente tampoco sabe con precisión la información "real".

Es importante realizar muchas pruebas con gente diferente para que comprueben si el resultado es realista. Algunas leyes de causa-efecto pueden ser contradictorias y darse al mismo tiempo. ¿Cuál de ellas utilizaremos entonces?, pues debemos utilizar la que sea aceptada por la mayoría de gente.

Por último, no intentemos apuntar muy alto. Mientras más real intentemos hacer un juego más posibilidades habrá de que consigamos precisamente lo contrario, debido a que tendremos demasiados aspectos del juego que controlar y es más fácil que nos olvidemos de algo por error. La mayoría de sistemas simples producen buenos resultados en cuanto a realismo.

Consiguiendo Jugabilidad

Lo primero que debemos hacer para que un juego sea jugable, es conseguir que tenga un sistema amistoso. Es decir, que las reglas y los componentes sean atractivos, fáciles de comprender y fáciles de usar.

La segunda regla y aún más importante es que debemos conseguir que sea fácil y atractivo el aprendizaje del sistema. Esto es lo que hará que nuestro juego enganche a la gente: lo que puede ser un juego simple para alguien entendido en el tema puede ser algo realmente incomprensible para un principiante. Solamente debemos tener en cuenta quién será el usuario final a la hora de diseñar el juego para saber de qué manera debemos presentar el sistema de aprendizaje.

Finalmente, debemos usar la lógica y los tópicos. Esto ayuda a un aprendizaje fácil, a una memorización sencilla y a una forma de jugar intuitiva. Además esto ayudará a que el juego parezca realista. Quizás en los juegos de rol encontremos la mayor concentración de lógica entre todos los géneros debido a que utiliza, en muchos casos, la lógica para diseñar las reglas.

Conclusión

¿Qué es lo que podemos sacar de todo esto? Pues que el realismo y la jugabilidad son importantes por igual. Lo que debemos conseguir siempre es el balance adecuado entre realismo y jugabilidad, dependiendo del género y del mercado que queramos tocar.

Debemos tener siempre presente que juegos poco realistas son muy jugables, y por tanto muy atractivos; y que, al contrario, algunos juegos muy realistas, si resultan injugables, pueden ser obviados por el grueso del mercado y pasados por alto después del enorme esfuerzo que puede haber ocasionado su diseño.

Por lo tanto os aconsejamos que intentéis que vuestros productos sean tan jugables como podáis y que parezcan lo más realistas posible. Podéis llevaros más de una sorpresa en cuanto realismo si lo intentáis continuamente: mientras más trabajes en un proyecto más realista será.

Jordi Martín Caballero



La ciencia ficción también tiene sus reglas de "realismo".

Posando para DIV (II)

Entramos en materia

Aunque resulta difícil de creer, habrá lectores que no habrán encontrado *FastTracker* de su agrado. Para ellos presentamos en este número una alternativa que poco tiene que envidiarle al programa que nos ha mantenido ocupados durante varias entregas.

En el artículo anterior vimos los menús y opciones básicos del Poser 2.0, programa que se incluyó gratuitamente en el CD de la revista. Ahora crearemos nuestra primera animación de personajes y, con ella, nuestra primera biblioteca de posturas.

En este número, vamos a intentar comprender las distintas opciones del programa, relacionadas en el artículo anterior, aplicando las más importantes a nuestra primera animación. Para ello, he elegido un problema clásico y de utilidad para cualquier videojuego: los movimientos de "carrera".

Casi todos los videojuegos en los que participan personajes bípedos deben representarles, en algún momento, corriendo. Tener que realizar un elevado número de sprites, para mostrar en pantalla adecuadamente esta acción, es una tarea ardua que requiere largo tiempo y que no siempre resulta todo lo real que quisiéramos. Pronto veremos que, con Poser, el problema está resuelto en apenas unos minutos.

El esquema de animación es una secuencia de posturas que da como resultado un movimiento

Documentación, documentación, documentación

Lo apuntábamos ya en el artículo anterior en el apartado de consejos: utilizad toda la información de la que podáis disponer para representar los movimientos humanos. Existen varias series de libros de fotografías de movimientos de personas, en las que se muestran secuencialmente todos los pasos que da el cuerpo para distintas actividades: correr, andar, saltar, etc. Estos libros son de gran utilidad para el desarrollo de videojuegos con personajes bípedos, pues nos sirven perfectamente como guías para calcar cada uno de los puntos clave del movimiento humano.

Si no podéis conseguir alguno de estos libros, no os preocupéis, también existe documentación sobre animación clásica (tipo Disney) bastante barata en cualquier librería; incluso en Internet podréis encontrar gran cantidad de páginas sobre animación, y, finalmente queda un último recurso: vosotros mismos y un espejo.

Representad las mismas acciones que vais a reproducir en el ordenador, delante de un espejo y fijándoos en cada parte del cuerpo que participa. Esto permite comprender perfectamente las relaciones entre todos los miembros a mover, muy útil para aplicar posteriormente cinemática inversa o, simplemente, dotar de realismo a cualquier movimiento (de paso, te permite darte cuenta de lo complicadas que son las leyes de la física: para cualquier movimiento del cuerpo hay que usar partes opuestas entre sí que le den impulso, como, por ejemplo, al saltar hacia adelante: primero estiramos los brazos hacia atrás para coger impulso y equilibrar el cuerpo, flexionamos las piernas para proporcionar la potencia necesaria, etc.).

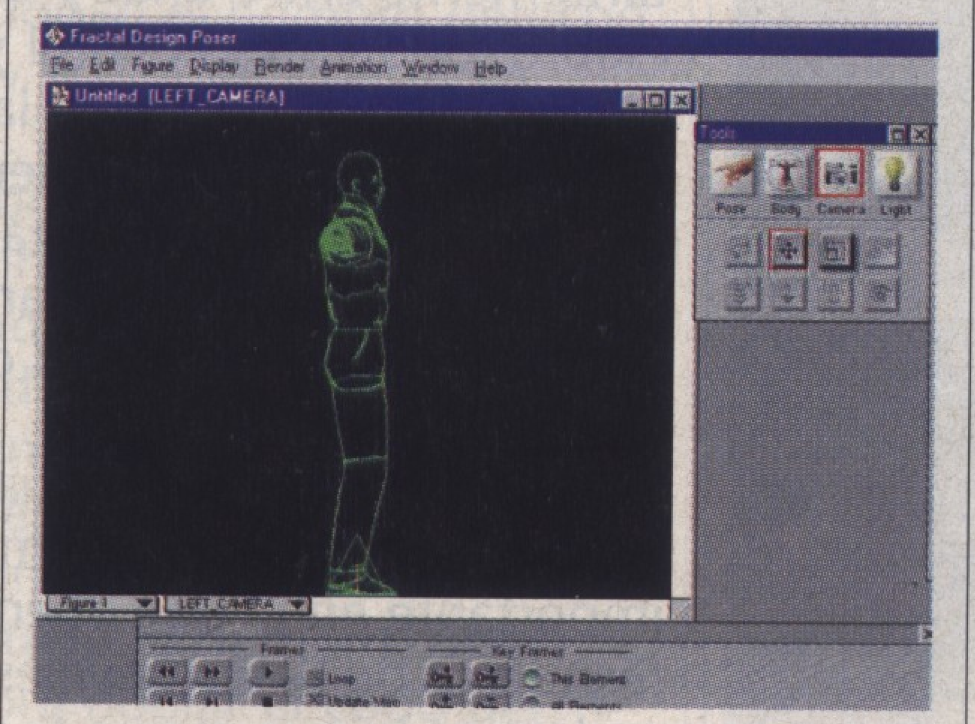


Fig. 2. Todo preparado para animar.



Fig. 1. Secuencia de movimientos del corredor, según Preston Blair.

Para el ejemplo he utilizado el esquema clásico de animación de carrera que dibujó Preston Blair hace años. Blair era un jefe de animadores de la Disney que participó en gran número de películas (Fantasía, Blancanieves,...) y fijó muchas de las reglas que, aún hoy, se siguen en la animación. Por ejemplo, determinó que casi todos los fonemas del habla se pueden resumir en doce expresiones diferentes de la boca del dibujo, estableció la diferenciación física típica de caracteres de cualquier película (el malo puede ser grande y con pinta de brutote o pequeñito y ágil, pero no de estatura normal) etc.

El esquema de animación que fijó (figura 1) es una secuencia cíclica de posturas que dan como resultado el movimiento de correr. Podemos ver que la secuencia no muestra el total de movimientos, ya que a partir de la postura 6 se repiten todos los gestos, pero con

los miembros opuestos (si en la figura 1 mueve hacia atrás el brazo derecho, en la 6 se mueve hacia atrás el izquierdo). Así, he copiado en la figura 1 el diagrama dos veces para que podáis seguir el ciclo normalmente. A la postura 10 seguiría de nuevo la 1.

¿Cómo animar?

Generalmente en los estudios de animación se realiza un estudio de la animación a dibujar, se discute y, posteriormente, el jefe de ese grupo de animación realiza las "animaciones claves". Éstas son los puntos básicos de la animación que definen por sí todo el movimiento. Por ejemplo, en el acto de saltar hacia adelante hay varios puntos clave: toma de impulso, salto, extensión en el aire, toma de tierra. El jefe de animación realizaría todos estos dibujos guía, dejando al resto de dibujantes la realización de los dibujos intermedios entre cada una de esas animaciones claves.

Nosotros no tenemos tanto personal que nos libere del trabajo sucio como los estudios de animación, pero a cambio tenemos Poser que nos va a echar una mano en todo este proceso.

Poser es capaz de realizar las animaciones intermedias

Con Poser también tenemos la opción de dibujar los puntos clave y dejar que el programa haga las animaciones intermedias, sin embargo, para este ejemplo haremos cada fotograma nosotros, pues así aprenderemos correctamente a desarrollar toda la animación. Una vez sepamos animar paso a paso, dibujar unas cuantas claves y que el programa haga el resto es pan comido. Además, las animaciones realizadas paso a paso tienen un aire más natural debido precisamente a pequeñas imperfecciones o defectos en los movimientos, que se dan en la vida real.

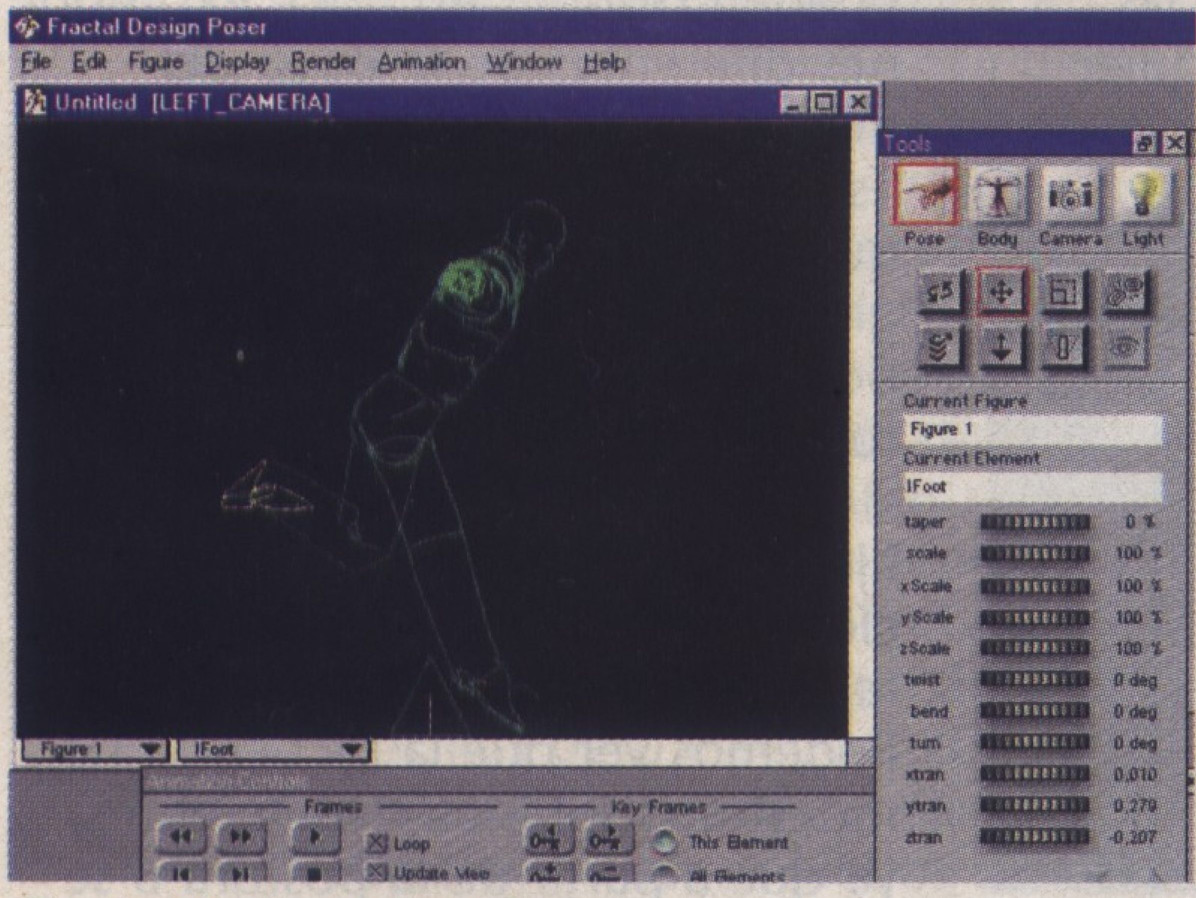


Fig. 3. Situando los pies.

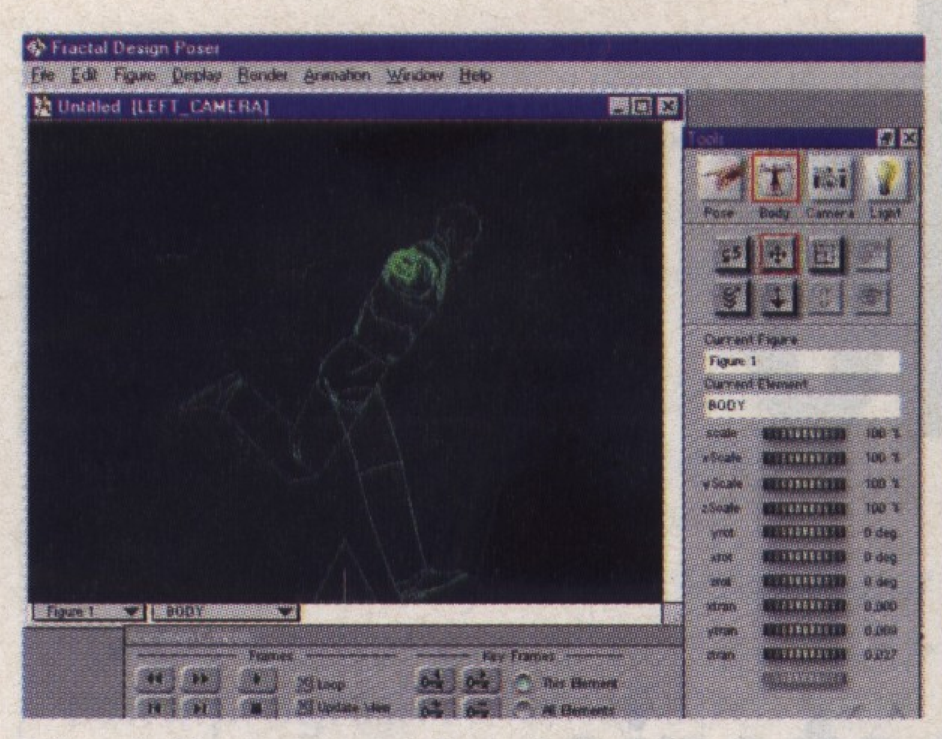


Fig. 4. Ya tenemos las piernas en su sitio.

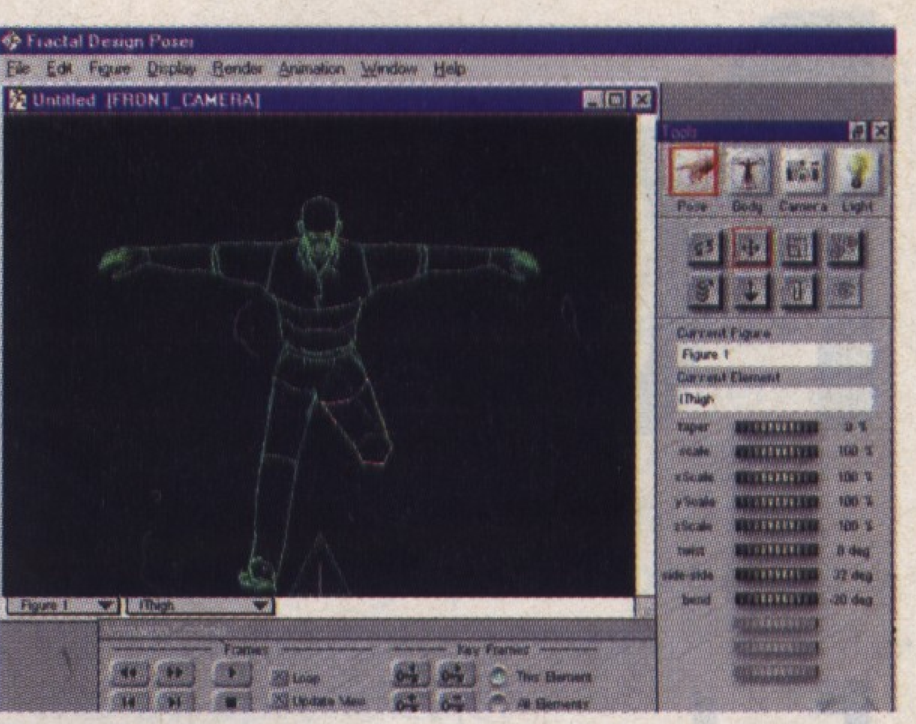


Fig. 5. Dándole vida al movimiento.

Todo el proceso no nos llevará demasiado tiempo, como veréis en unos instantes, y pronto olvidaréis esas largas horas con el editor de sprites de Div intentando hacer correr a vuestro personaje.

Nuestro personaje empieza a correr

Lo primero que debemos hacer es insertar un objeto que nos servirá de guía para saber siempre dónde están situadas las caderas del personaje. Para ello, abrimos el menú *Display/Add Prop/Cone*, con lo que nos aparecerá un cono en el suelo, justo en el centro de la imagen, señalando la situación inicial del personaje.

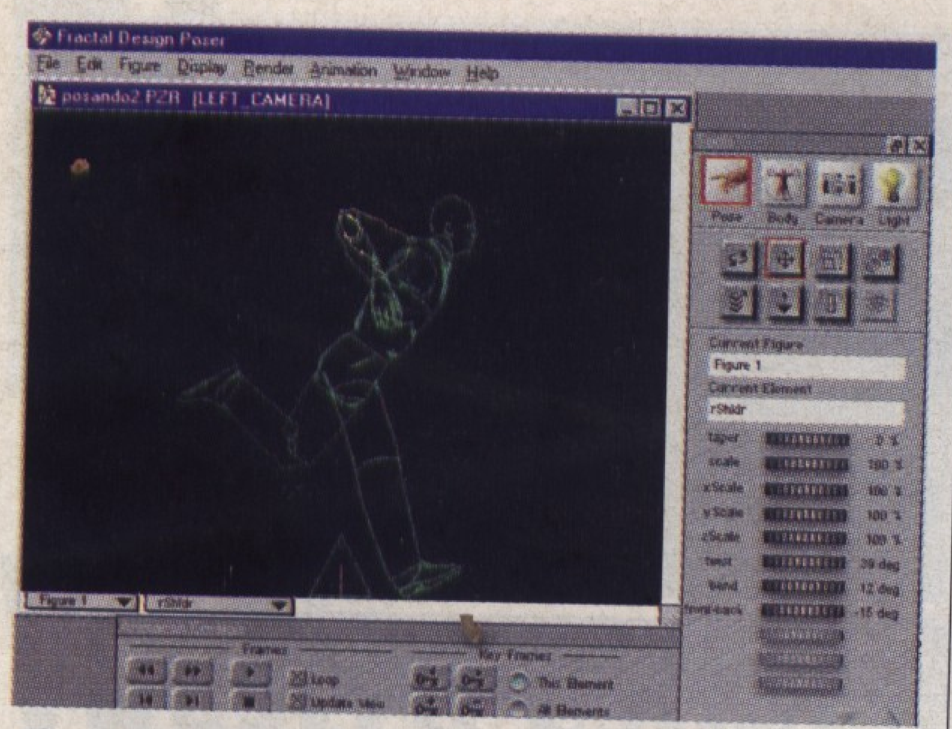
También es conveniente activar la opción *Display/Depth_Cued*, que nos muestra en colores más brillantes aquellas partes del cuerpo que están más cercanas al espectador en esa vista, es decir, si estamos mirando el perfil derecho de una figura, la pierna derecha aparecerá de un verde brillante, y la izquierda en un verde pálido, indicando que la pierna derecha está por delante de la izquierda desde nuestro punto de vista.

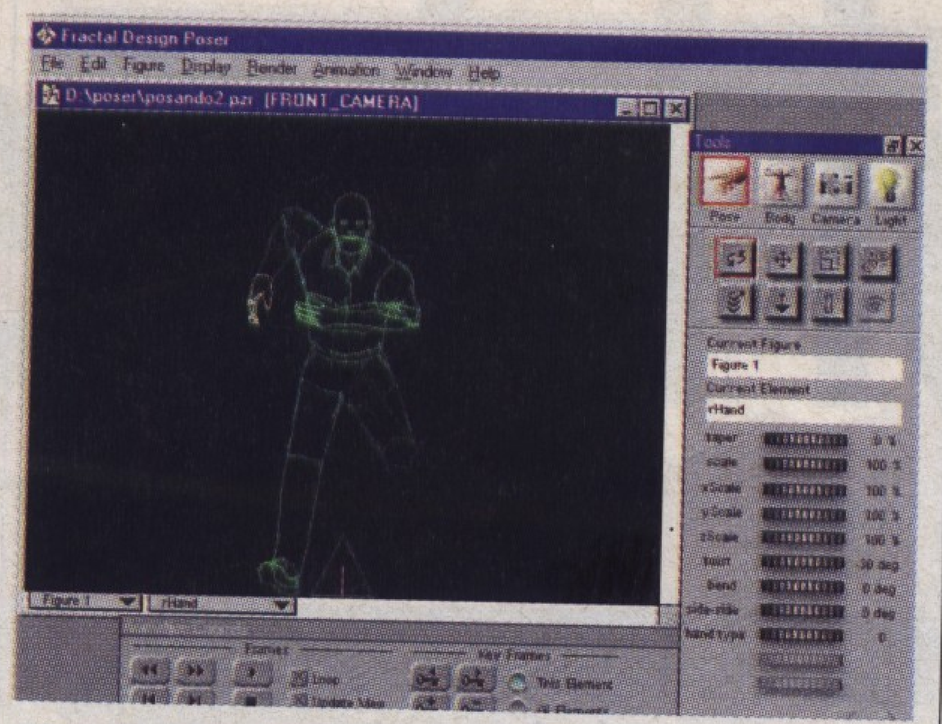
Para realizar correctamente la animación, es fundamental moverse entre las tres cámaras básicas en cualquier representación 3D: Perfil, Frente y Planta. Estas vistas se corresponden con la *Left_Camera*, *Front_Camera* y *Top_Camera*, cuyas teclas de acceso rápido son *Ctrl-L*, *Ctrl-F* y *Ctrl-T*. Es mejor usar estos atajos de teclado que acudir al menú *Display/Camera_View/Left_Camera*, ya que necesitamos cambiar constantemente entre unas y otras (sobre todo entre el perfil y el frente) para ver como varían las posiciones de los distintos miembros de la figura. Para comenzar, pulsaremos *Ctrl-L* para visualizar correctamente el personaje y el cono-guía (fig.2).

En la animación de cualquier personaje hay un punto muy importante en el que apenas se fijan los dibujantes noveles: la cadera es la que marca todo el movimiento del esqueleto. Situándola correctamente tenemos hecho gran parte del trabajo posterior. Para ello, observemos el esquema de Blair. Podemos ver que nuestra figura en su movimiento 1 está inclinada hacia delante (en realidad en toda la animación, pues es algo intrínseco al hecho de correr). Por lo tanto, procederemos a inclinar el torso de la figura. ¿Cómo inclinamos el tronco? Sencillo, la cadera se inclina para mover toda la parte superior del cuerpo a la vez. Pulsamos la lengüeta inferior de la ventana de la figura y en el menú desplegable que se presenta seleccionamos *hip* (cadera). La cadera de la figura aparecerá resaltada en blanco, indicando que ha sido seleccionada. Seleccionamos el icono *Pose* de la paleta *Tools* (el que tiene una mano dibujada), que indica que moveremos una parte del cuerpo y no el cuerpo entero. En la paleta de controles de la figura (la ventana con "rodillos"; si no la tenéis abierta seleccionad el menú *Window/Parameter Dials*) pulsad el rodillo *Bend* con el botón izquierdo del ratón, y moviendo éste a la derecha mantenedlo pulsado hasta que aparezca el número 36. Veréis, en la ventana principal, como la figura comienza a inclinarse.

Comencemos a situar las piernas de la figura, tal y como aparecen en el dibujo 1 de la secuencia de Preston Blair (fig. 1). Para ello, pulsamos la lengüeta inferior de la ventana de la figura, y seleccionamos *rfoot* en el menú. Comprobad que *Pose* sigue seleccionado en la paleta de herramientas y seleccionad el icono que muestra cuatro flechas hacia los puntos cardinales (*Mover*). En este caso, no utilizaremos los rodillos, puesto que la cinemática inversa que va asociada a los pies de la figura nos permite moverlos "a mano" con cierta seguridad. Para el resto de miembros, movedlos siempre con los rodillos.

Observando el dibujo en el que basamos todo el proceso, podemos ver que la pierna derecha está adelantada, con el pie tocando el


Fig. 6. Brazo derecho terminado.


Fig. 7. Nuestro personaje casi finalizado.

suelo con su talón, mientras que la pierna izquierda se recoge flexionada. Hagámoslo; una vez que hemos pulsado en el icono de *Mover*, movamos el pie derecho hacia delante, procurando mantener la pierna derecha estirada. Seleccionemos *lfoot* en el menú correspondiente y movamos el pie izquierdo hacia atrás y arriba. Probablemente la situación a la que habéis llegado se corresponde con la figura 3.

¿Qué vemos raro en esta figura? Los pies están retorcidos, el personaje está en el aire, la cadera está desplazada respecto al cono... un desastre, vamos.

Comencemos por arreglar el pie izquierdo. El pie está bien situado pero torsionado respecto a su postura natural. Movamos el rodillo *Bend* a la derecha con dicho pie seleccionado hasta el valor 149. Ya está arreglado.

Seleccionemos ahora la cadera de nuevo (*hip*) y, con la herramienta de mover de nuevo, desplacémosla hacia abajo, centrándola al mismo tiempo sobre el cono.

Corrijamos ahora el pie derecho. Dado que debe estar posado sobre su talón, sólo se trata de aplicar *Bend* una vez seleccionado ese pie, hasta que veamos que la puntera se levanta y sólo se apoya la figura sobre el talón derecho. El personaje ahora debería parecerse a la figura 4.

Si acudimos a la vista frontal (*Ctrl-F*) podemos ver que todo está situado correctamente, pero es demasiado "artificial". Las piernas están completamente paralelas, dando sensación de rigidez. Los robots probablemente correrán así algún día, pero no nosotros.

Vamos a dar vida a estas piernas. Seleccionad el pie derecho desde la vista frontal y desplazadlo con la herramienta *Mover*, a la izquierda de la pantalla, separándolo ligeramente del cono. Ahora seleccionad el muslo izquierdo de la figura (*lThigh*) y girando los rodillos aplicadle *Side-side* (mueve el muslo de lado a lado de la imagen) hasta el grado 32. Ahora, el personaje parecerá estar corriendo desenfrenadamente (o al menos sus piernas lo parecen), como en la figura 5.

Y, tras las piernas, el resto del cuerpo

Si observamos detenidamente los dibujos de Blair, o incluso nuestro propio cuerpo, cuando corremos, veremos que aunque el tronco esté inclinado nuestra cabeza sigue erguida mirando al frente.

Levantemos la cabeza del personaje: volvamos a la vista de perfil (*Ctrl-L*), seleccionad la cabeza (*head* en la pestaña correspondiente de la ventana principal), y aplicadle al rodillo *Bend* –33 grados; esta vez el valor es negativo, para lo que hay que mover el ratón hacia la izquierda, en vez de a la derecha como en las ocasiones anteriores. Veremos como nuestro personaje yergue la cabeza, mirando altivamente al frente.

Para el braceo debemos tener en cuenta un aspecto muy importante: los hombros. Éstos marcan todo el movimiento de los brazos, por lo que si un brazo se inclina hacia atrás es porque su hombro correspondiente le obliga a ello.

Otro aspecto igualmente importante es que el brazo que se mueve hacia delante es el opuesto a la pierna que lo hace en esa dirección. Observemos nuestro gráfico guía: la pierna derecha es la que se adelanta en esta fase de la carrera, luego será el brazo izquierdo el que se mueva vigorosamente hacia delante. Por el contrario, el brazo y la pierna izquierda se desplazan hacia atrás.

Seleccionad el hombro derecho (*rCollar*), y aplicadle *Front-Back* hasta –45. Lo que hemos hecho es rotar el hombro derecho hacia atrás 45 grados, para continuar modificando el brazo.

Aplicad ahora *Twist* a *rShldr* hasta alcanzar 39 grados. Ahora aplicad a este mismo miembro *Bend* 12 grados y *Front-Back* –15. Hemos rotado el hombro, para poder ahora doblar el codo del brazo derecho hacia abajo.

Es hora de mover el antebrazo derecho: seleccionadlo (*rForeArm*) y aplicad *Bend* 116 grados. Si variáis entre la vista frontal y perfil veréis que el brazo va quedando conforme a la postura prevista en la guía.

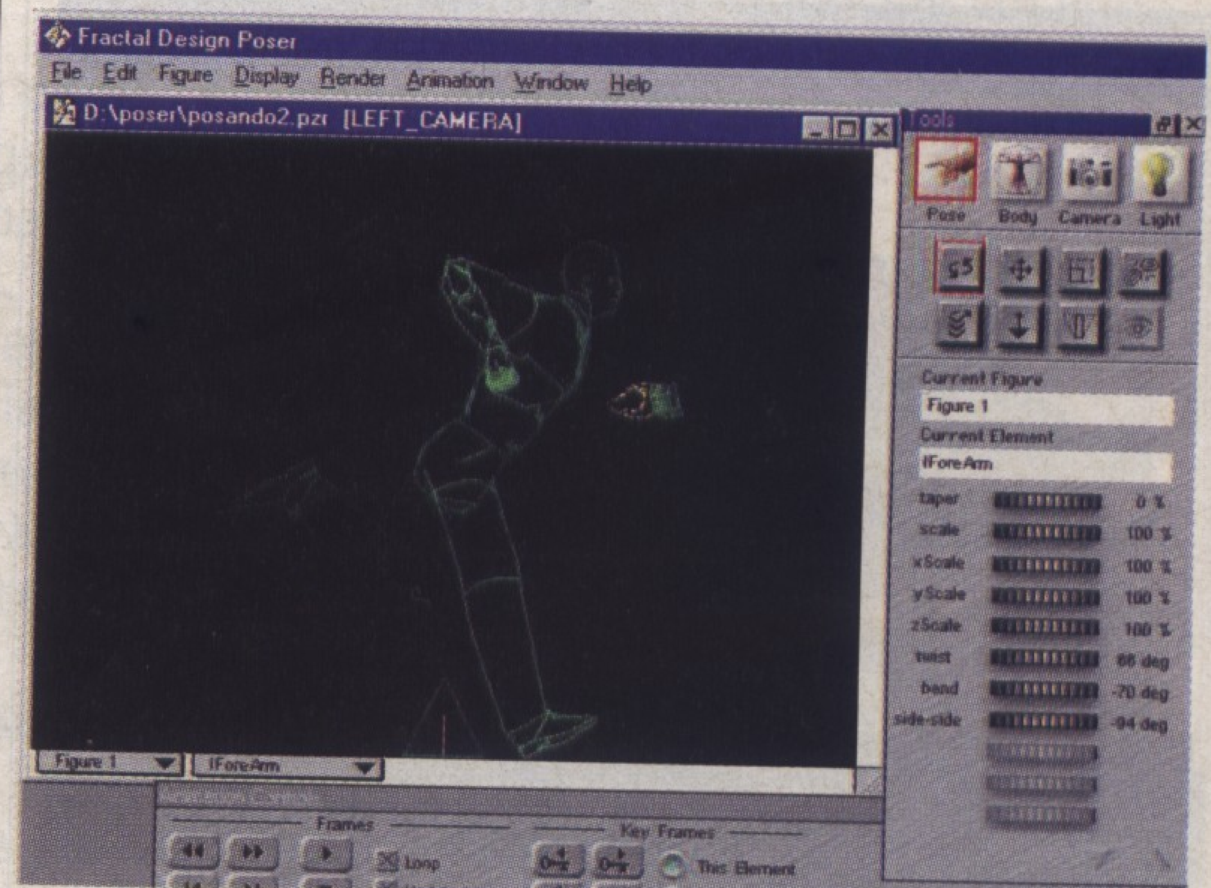
Para dejar la mano derecha (*rHand*) en posición, sólo hay que rotarla 116 grados con *Twist*. ¡Ya tenemos el brazo derecho en pleno braceo! (Figura 6). Vamos a por el izquierdo.

En realidad esto comienza a ser casi rutinario: sólo se trata de seleccionar el miembro a mover, y con los rodillos correspondientes aplicarle algún tipo de torsión, que se reflejará instantáneamente en la figura de la ventana principal. Así, si vemos que algún movimiento no es el previsto o si, simplemente, no nos gusta cómo está quedando, tenemos dos opciones: volver a situar el rodillo alterado en el valor 0, o pulsar *Ctrl-Z* (*deshacer*) para volver a la postura anterior del miembro modificado.

Seleccionamos el hombro izquierdo (*lCollar*), y le aplicamos con los rodillos *Front-Back* hasta –28 grados. Esto hará que dicho hombro gire ligeramente hacia delante. Podemos ahora aplicar a *lShldr: Twist* –29, *Bend* –2 y *Front-Back* –46 y a *lForeArm*: un *Bend* –70 y *Side to Side* –94, que dejará el brazo izquierdo finalmente como queremos.

La cadera es la que marca el movimiento del resto del cuerpo

Nos falta un último detalle (Figura 7): nuestro personaje está corriendo con las manos extendidas y, probablemente, el efecto de carrera se vería incrementado si le ponemos las manos en forma de puño. En realidad, esta opción deberíamos haberla seleccionado antes de comenzar a animar la figura, pero la he dejado para este momento para que podáis ver cómo es posible variar cualquier miembro del cuerpo a lo largo de la animación. Para ello, sólo es


Fig. 8. Figura terminada.

necesario seleccionar primero una mano y hacer un doble clic con el ratón sobre el rodillo de *Hand-Type* (una forma rápida de introducir los valores que deseamos en los rodillos, sin necesidad de girarlos con el ratón). En la ventana que nos aparece sólo hay que introducir el valor 14 en el campo *Value* para que veamos transformarse la mano extendida en un puño. Repetid el proceso con la otra mano y la primera figura ya está completa.

Retoquemos ahora algún detalle suelto como, por ejemplo, la mano izquierda, que debería estar torsionada un poco más hacia arriba (aplicad *Twist* a *lForeArm*, hasta 66 grados). En la Figura 8 podéis observar como queda una primera aproximación del personaje.

Como veis, sólo lleva unos minutos conseguir un personaje que antes nos obligaría a emplear horas en dibujar en DIV o, simplemente, que no se podría hacer en el *Generador* de *Sprites* en caso de tratarse de una postura complicada.

Creemos nuestra propia biblioteca de posturas

El proceso para seguir realizando el resto de la animación es simple, sólo hay que variar la postura que hemos conseguido aquí, para pasar a la postura 2 del diagrama de Blair, luego a la 3, etc. Pequeñas variaciones en los pies, brazos y cadera nos permiten crear todos los movimientos rápidamente.

Con este programa es posible variar cualquier miembro del cuerpo

Debéis tener en cuenta que (tal y como muestra la línea punteada del dibujo de Blair) en el movimiento de carrera la cabeza no siempre está a la misma altura: cuando los pies despegan del suelo la cabeza alcanza su punto más alto y cuando las piernas se flexionan tras volver a tomar tierra la cabeza llega a su punto más bajo. Estos detalles son los que dan realismo a los movimientos y deben ser tenidos en cuenta en las animaciones.



Fig. 9. Comprobad la postura desde todas las vistas.

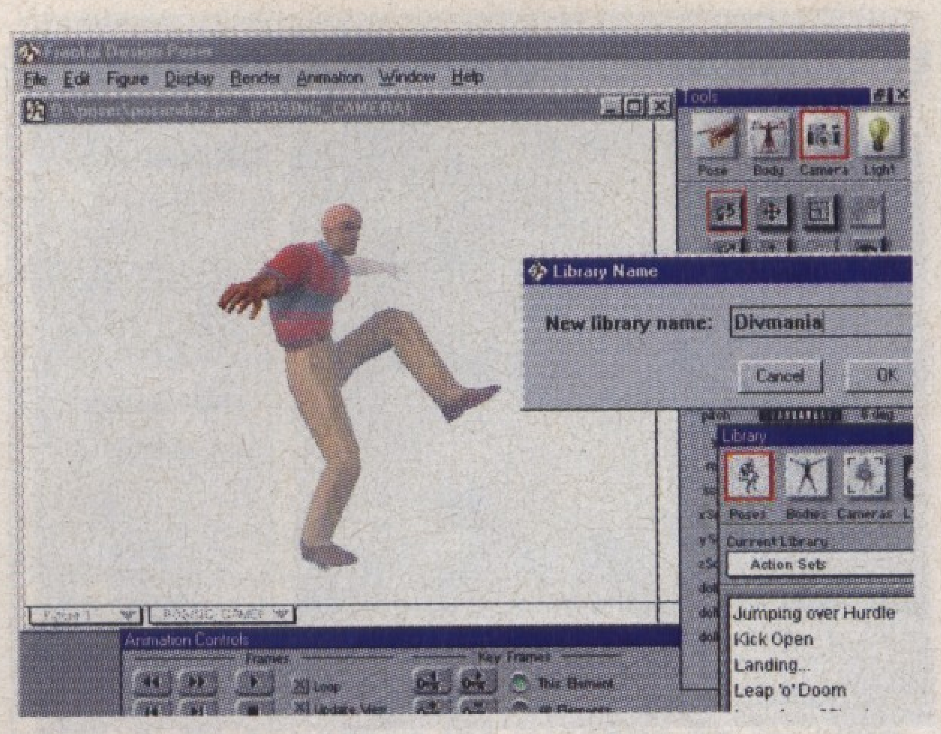


Fig. 10. Asignándole nombre a nuestra biblioteca.

Para poder crear varias posturas distintas para formar la animación, sabiendo que cada postura será una variación de la anterior, la mejor opción es crear nuestra propia biblioteca de posturas. Si seleccionáis el menú *Windows* y, dentro de él, marcáis la opción *Library* os aparecerá una nueva paleta (en el caso de que no la tuvieseis ya seleccionada) en la que podréis encontrar las posturas, cuerpos, cámaras y luces creados por el equipo de Metacreations, agrupadas por distintos tipos (por ejemplo, las posturas se agrupan por "Grupo de Deporte" (*Sport Sets*), "Grupo de Baile" (*Dance Sets*), etc. bajo el desplegable de *Current Library*). Dentro de cada grupo se encuentran las posturas que lo integran, que se visualizan en la lista que ocupa la mayor parte de la paleta.

Ahora vamos a crear nuestra propia biblioteca, o grupo de posturas, en esa paleta y a meter en ella las posturas que vayamos creando. Para ello pulsamos el desplegable de *Current Library*, y observaremos como la última opción de la lista que se nos muestra es *Create new*. Seleccionamos esta opción y se nos mostrará una ventana que nos solicita que introduzcamos el nombre de la nueva biblioteca de posturas a crear. Introducid el nombre que creáis conveniente, nosotros hemos puesto ***Divmanía*** (Figura 10), pero podéis darle un nombre descriptivo que os indique qué estáis guardando en ese grupo de posturas, como "Carrera", "Saltos", "Lucha", "Ataques con lanza" o lo que mejor represente los movimientos que guardaréis en esa librería.

Una vez creada esta librería aparecerá en la última posición de la lista desplegable que se muestra en *Current Library*, permitiéndonos seleccionarla en cualquier momento.

A continuación podemos añadir la postura que hemos creado en la primera parte de este artículo, de forma que podamos asignar ese movimiento a cualquier figura que seleccionemos en otro momento. La forma de hacerlo es bien sencilla: si seleccionamos la biblioteca que acabamos de crear, veremos que la lista de posturas que contiene está vacía. En la parte inferior de la paleta *Libraries* podemos comprobar que existen tres botones que nos permitirán incluir nuestras propias posturas en la biblioteca recién creada. Estos botones son *Use* (permite aplicar a cualquier figura la postura que tengamos seleccionada en ese momento en la biblioteca), *Add* (añadir postura a la biblioteca) y *Delete* (eliminar postura de la biblioteca).

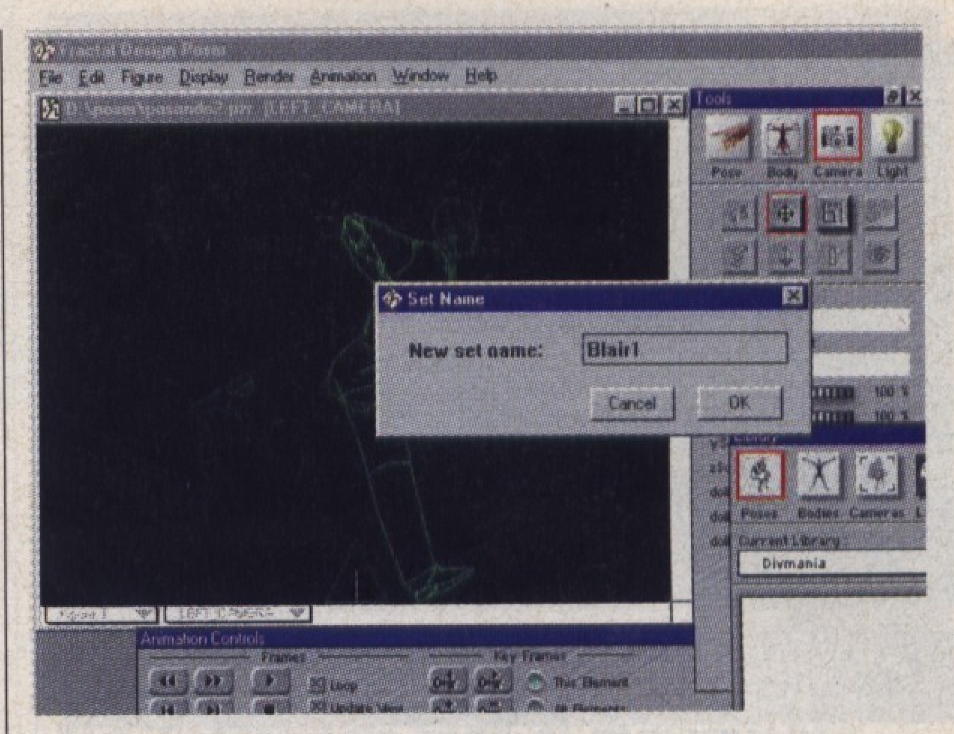


Fig. 11. Hay que nombrar cada postura.

Así, con la postura que quedamos añadir en la pantalla principal, pulsemos el botón *Add* de la paleta de *Libraries*; en la ventana que nos aparece solicitándonos un nombre para ese *set*, introducid el nombre que describa esa postura en concreto (Figura 11 – hemos introducido "Blair1") y posteriormente se os solicitará que seleccionéis si deseáis grabar una animación completa o una sola postura. En próximos artículos veremos cómo guardar animaciones completas, así que de momento escoged la opción *Single Frame* y aceptar.

En el próximo artículo veremos cómo crear la animación definitiva partiendo de varias posturas estáticas y aprenderemos también a crear nuestras propias texturas para nuestros personajes. Sobre este último aspecto, hemos recibido varias consultas por lo que os invito a reencontrarnos dentro de dos meses para que podáis comprobar lo sencillo que resulta "vestir" nuestras figuras con la indumentaria que más nos guste. Mientras tanto, os recuerdo que podéis plantear vuestras dudas en la dirección de correo electrónico *tayete@teleline.es.*

Santiago García Mazariegos "Tayete"

El correo del lector

Vuestro espacio

Todas las dudas que podáis tener acerca de la programación de juegos serán raudamente contestadas en este espacio. Así que no lo dudéis, explicarnos vuestras dificultades y pronto dejarán de serlo. También recogemos, como es ya habitual, todas aquellas ofertas de trabajo que puedan interesar a todos los que estén inmersos en el mundo de la programación de videojuegos.

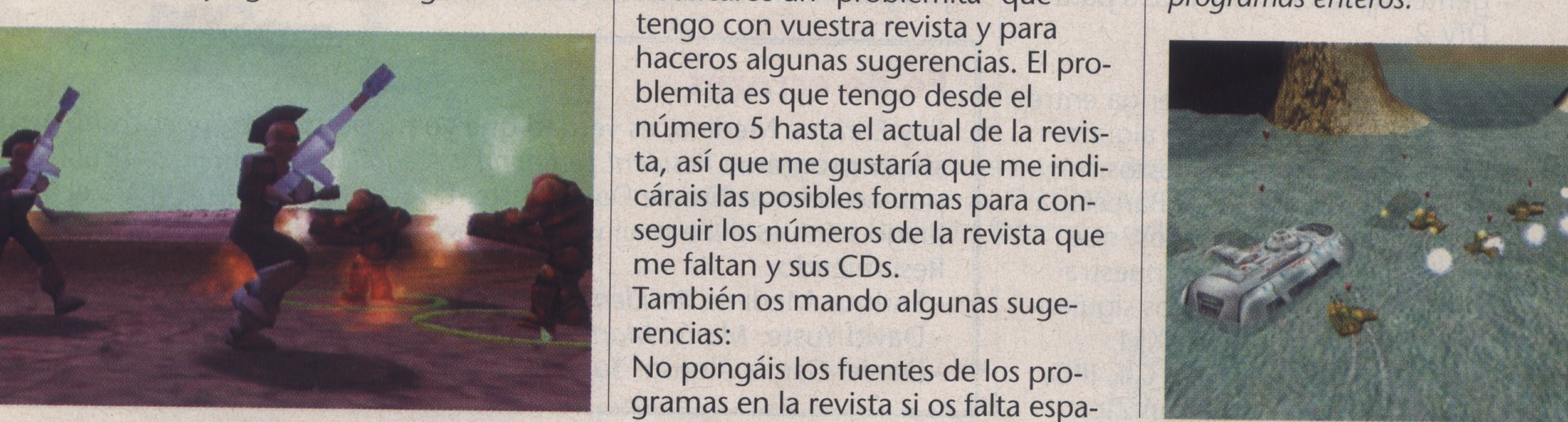
Gente de DIVmanía: gracias por hacer DIVmanía, me hace sentir con apoyo en este apasionante pero difícil mundo de la programación de videojuegos y es por ello que creo que Uds. como responsables de nuestra publicación tienen la complicada tarea de "satisfacernos". Me explico: todos sabemos que programar con DIV es fácil, pero ¿qué pasa con los que ponemos empeño e ilusión en nuestros trabajos y que a pesar de todo quedamos en medio? Creo que los cursos de programación (estrategia, rpg, etc..) y muchas secciones de la revista se quedan a mitad del camino, por la sencilla razón que hay términos y conceptos que muchos no dominamos y se avanza en ellos siempre superficialmente. Creo que se debería prestar más atención a los lectores que buscamos en DIVmanía una fuente de aprendizaje, ya que es nuestro único medio de contacto con gente que sabe del tema, por eso ahí van unas ideas que creo satisfacerían a muchos de nosotros:
Se podría sacar a la calle un "especial" o algo así, p. ej. "Cómo programar un juego de estrategia con DIV paso a paso" y que en él se comentara y explicara desde el "program" hasta el último "end" con más detalle que en los artículos. Otra idea sería crear una nueva sección donde se analizaran pequeños trozos de código que supusieran soluciones a problemas usuales, por ejemplo: ¿cómo hacer para que el scroll que sigue a un proceso dejara de hacerlo al llegar al fin del mapa? o ¿qué hay que hacer para que desaparezca un disparo al chocar con un enemigo?.. y cosillas así.
Mil gracias por adelantado. Ariel A. Álvarez

Tomamos nota de tus sugerencias, intentamos complacer a todos pero es muy difícil ya que los que se dedican al noble arte de la programación de juegos tienen diferente nivel de conocimientos. Te recomendamos que si tienes una duda con respecto a cualquier artículo te dirijas al autor del mismo vía correo electrónico, estarán encantados de ayudarte.

Cómo conseguir números atrasados

¡Hola, DIVmanía!, os escribo para notificaros un "problemita" que tengo con vuestra revista y para haceros algunas sugerencias. El problema es que tengo desde el número 5 hasta el actual de la revista, así que me gustaría que me indicárais las posibles formas para conseguir los números de la revista que me faltan y sus CDs.
También os mando algunas sugerencias:
No pongáis los fuentes de los programas en la revista si os falta espacio para otras cosas, ponerlos en el CD. Tendréis más espacio.
No pongáis tantas capturas en la revista, algunas parecen que están para hacer bulto, algunas menos pero bien elegidas estaría bien.
En la sección de Utilidades, poned cosas más relacionadas con la creación de juegos. Ignoro si es un caso aislado, pero lo de *FinePrint 3.60* (número 5) me pareció fuera de lugar y creo que en la Red hay programas share o freeware bastante interesantes como para dedicarles la sección, como *Fénix* (el programa de Cebrián que apareció en el número 7); por contra, el *FreeMem Pro v4.2* (número 4) y el *Audiograbber 1.61* (número 7) me parecieron más acertados. Pero lo del *FinePrint 3.60* ... (suspiro).
¿Por qué en el número 7 ponéis Poser 2 y Bryce 2 y el nº de teléfono para conseguir los números de serie?. Haberlos puesto directamente, ¿no?
Rafael José Navarro Molina

Resolvemos tu duda diciéndote que para conseguir números atrasados de Divmanía deberás ponerte en contacto con el departamento de suscripciones, es la misma dirección y telefono que aparece en el cuadro, sólo sustituye Divmanía por Suscripciones y ellos se encargarán de mandarte los números atrasados que quieras. En cuanto a tus sugerencias, tomamos nota, como siempre, y en lo referente a que teníais que llamar por teléfono para conseguir los códigos de Poser 2 y Bryce 2, pues deciros que eso era uno de los requisitos indispensables que nos puso Metacreations para dar los programas enteros.





Se busca gente

Hola. Lo primero daros las gracias por darnos la oportunidad de darnos a conocer ante todos los DIVmaniacos/as. Me llamo Alberto y tengo 17 años, me gustaría ponerme en contacto con gente que le interese entrar en un ambicioso proyecto que tengo en la cabeza. Necesito tanto programadores/as como grafistas 2d. La edad no me importa mucho. El proyecto es un juego de scroll horizontal, de ambiente medieval, tipo *Golden Axe*. Aquellos que estén interesados que me escriban un e-mail a la siguiente dirección:
albertoblanc@yahoo.es.
Muchas gracias a todos y seguid así.

Un desarrollador agradecido

Hola, soy Germán Bermejo el autor de *99-00*. Os mando este correo para agradeceros el tratamiento que le habéis dado a mi juego. Me parece que algunos calificativos han sido muy generosos. Desde hace un par de semanas estoy conectado a Internet y he estado curioseando por las revistas de juegos, me he dado cuenta que el género de las aventuras está un poco muerto, pero no desespero, seguro que se ponen de nuevo de moda.

Un anuncio

Nothing Software, grupo de creación de Videojuegos, busca:
– grafistas 3D y/o 2D.
– músico o creador de FX sonoros.
– programadores de DIV 1 ó 2.
– gente capaz de crear DLL´s para DIV 2.
– creadores de Webs.

Necesitamos gente que tenga entre 14 y 22 años que domine alguno de los campos aquí expuestos. Preferencia residentes en Barcelona o provincia. Los interesados, que manden sus datos y una muestra de su trabajo en uno de los siguientes formatos: 3DS, MAX, XM, MOD, S3M, MP3, HTML, GIF, JPG, DLL, y que lo incluya en un Zip. Importante, poner como asunto: "creación de Juegos".
- Morpheo -
" Dani Santeugini" (Barcelona)
danis@apabcn.ictnet.es

Un error frecuente

Hola, necesito que me confirméis un error muy raro que tiene DIV2. El error es que, cuando creas una variable, o una tabla, o una estructura ,y le das el valor 'CON' , cuando lo compilas se bloquea. No importa cuándo le des el valor, ni dónde. Ni tampoco si está en minúsculas o en mayúsculas, ni el sistema operativo (windows o dos). Se bloquea siempre. Me gustaría que me lo confirmaseis ya que es muy raro... Gracias.
José Luis

Se ha hablado mucho de esta duda, tanto en la lista sobre DIV como en el canal de IRC, y es un misterio, una especie de bug, donde tiene algo que ver Windows 98, al parecer tiene problemas con la palabra "con", que servía antes para designar cosas de teclado en DOS, por ahora no se ha encontrado solución a este error.

Requerimientos de Ram

Me gustaría saber cómo calcular los requerimientos de RAM para ejecutar un determinado programa realizado con DIV o DIV2.

Es muy simple. Solo tenías que haberte mirado las funciones de manejo de memoria dinamica. Para saber cuanta RAM usa un programa en un determinado momento, solo tienes que tener una variable "memoria" a la que le asignes así:
memoria=memory_free()
En ese momento "memoria" contendrá, en Kbs, la memoria libre del momento en que se ejecuto la asignacion. Si multiplicas eso por 1024 memoria=1024; lo tendras en bytes. Si tienes la memoria libre y la memoria total de tu equipo, puedes hallar la usada fácilmente con una resta.*

Y para terminar

Soy un nuevo lector de la revista y estoy especialmente interesado en la programación de DirectX (3d).



El problema es que no sé por dónde empezar. Al principio iba a empezar por Open GL pero me convencieron de que DirectX tenía más futuro y estaba más extendido. He intentado buscar por la red algún curso en español, pero el esfuérzo ha sido inútil. Lo único que encontré fue un curso de OpenGL en www.macedónia.com, realmente bueno. Te escribo esto para ver si me podrías aconsejar cómo empezar en esto de la programación gráfica en 3D. Mis conocimientos de programación son C y Visual Basic y tengo el TurboC, el Visual Basic 5.0 y el Visual C++ 4.1, aparte del SDK de Direct X (creo que el 5.0).

Necesitas un buen libro. Te recomendamos alguno de Visual Basic 6.0 ó "Programación multimedia avanzada con DirectX", de Sánchez Ballesteros, Constantino, Editorial Rama; otro que te puede servir es "A fondo DirectX", éste te dará una idea de cómo programar con estas librerías. Luego debes documentarte tú, aparte, sobre 3D.

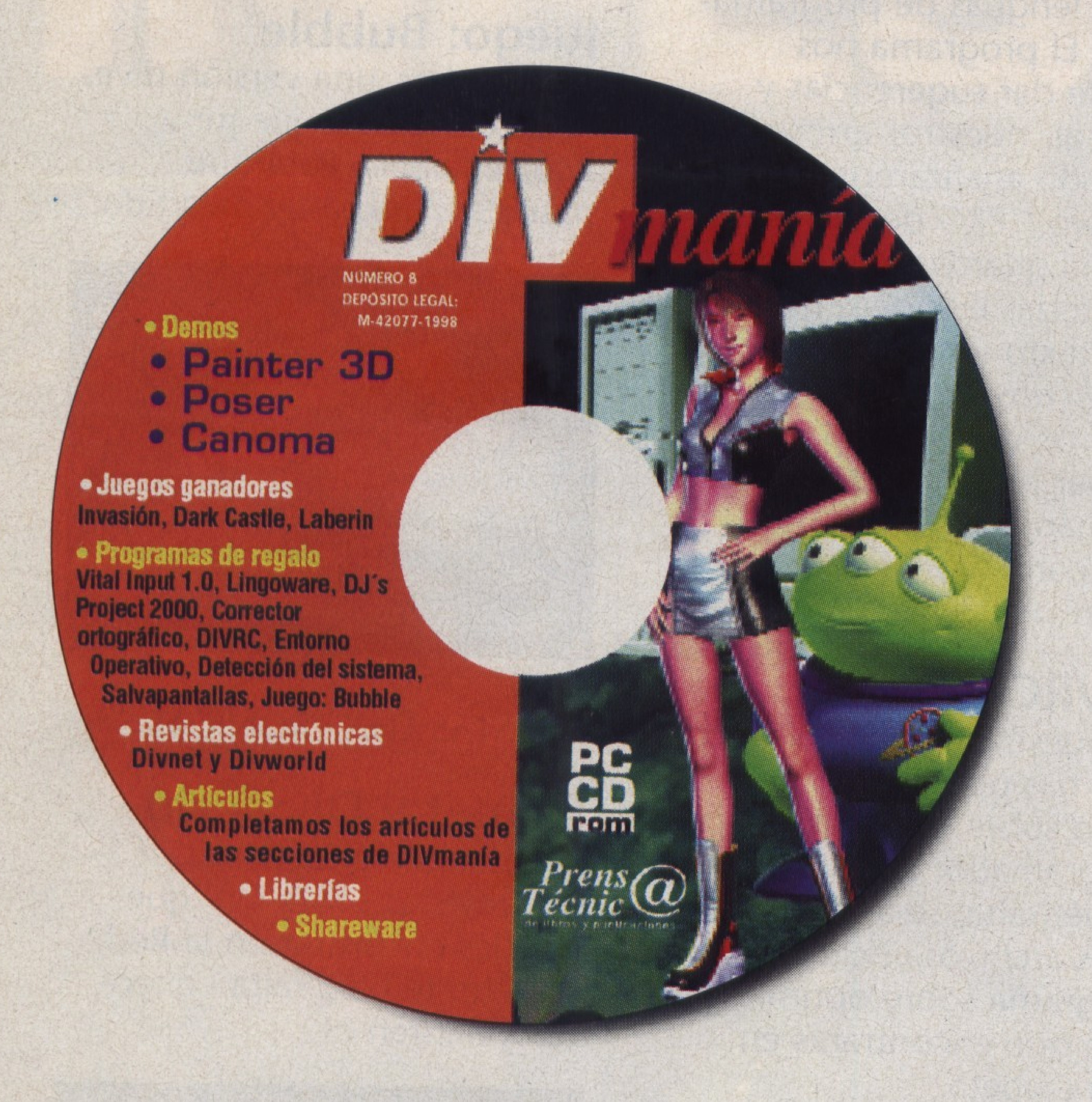
Esperamos vuestros mensajes

Para cualquier sugerencia, duda o aclaración podéis mandar vuestras cartas o correos electrónicos a la siguiente dirección:
Divmanía
C/ Alfonso Gómez 42. Nave 1-1-2
Madrid 28037. España.
Tfno: 91. 304. 06. 22
Fax: 91. 304. 17. 97
E-mail: *divmania@prensatecnica.com*

Fe de errores

Soy Enrique Medina, es verdad que yo no programé *Super Yoshi*, pero si que programe *Resident Lemming* y *Arena (Worms Arena)*, al cual le pusisteis el nombre de David Yuste. David fue el que me envió el juego a través del e-mail y posiblemente es confundísteis por eso.
Resumiendo:
- Enrique Medina: *Resident Lemming, Worms Arena.*
- David Yuste: *Misión Marte.*
- Daniel García Alonso: *Super Yoshy.*

Contenido CD-Rom



Este número ocho llega con un CD repleto de programas y utilidades para que los uséis en vuestros trabajos de programación, además, como ya es habitual, os ofrecemos los juegos ganadores de este mes. Esto es lo que podréis encontrar:

JUEGOS GANADORES

¡Tachán, tachán! ...y los ganadores de este mes han sido:

Invasion

Un arcade en el que tendremos que invadir la Luna como primer paso para acabar con la Tierra y de paso con los problemas de más de uno.

Dark Castle

Sobrevive a este laberinto repleto de peligros y logra encontrar la salida de tan lúgrube lugar, cuidado con sus habitantes.

Laberyn

Mata todo lo que encuentres y recoge todas las llaves para abrir las puertas y pasar de pantalla, algunas de ellas tienen hasta siete cerraduras.

DEMOS

Demo Painter 3D

Este programa os facilitará la labor de texturizar vuestros objetos en tres dimensiones, lucirán con vida propia.

Demo Poser 4

Programa demostración de lo que es capaz de conseguir esta herramienta de creación y animación de personajes, un complemento a la nueva sección de la revista basada en *Poser*.

Demo Canoma

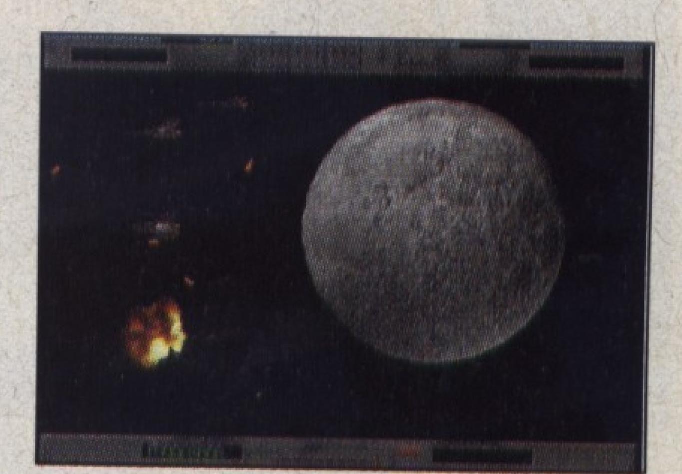
La utilidad de *Canoma* es convertir una simple foto en algo tridimensional. Seguro que le encontráis utilidad para el diseño de los escenarios de vuestros juegos.

PROGRAMAS

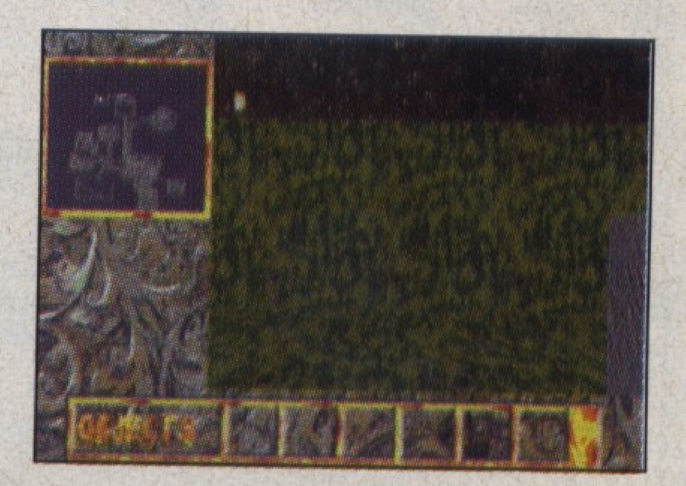
A nuestra redacción llegan de vez en cuando software creado por nuestros lectores que lo ofrecen de forma altruista para que todo el mundo pueda disponer de él y usarlo en su propio beneficio, al tiempo que es una forma de darse a conocer. En ocasiones se trata de juegos que no se presentan al concurso (por un motivo u otro) y en otras útiles programas que seguro muchos encontrarán la forma de aprovechar. Este mes tenemos unos cuantos programas que ofreceros.

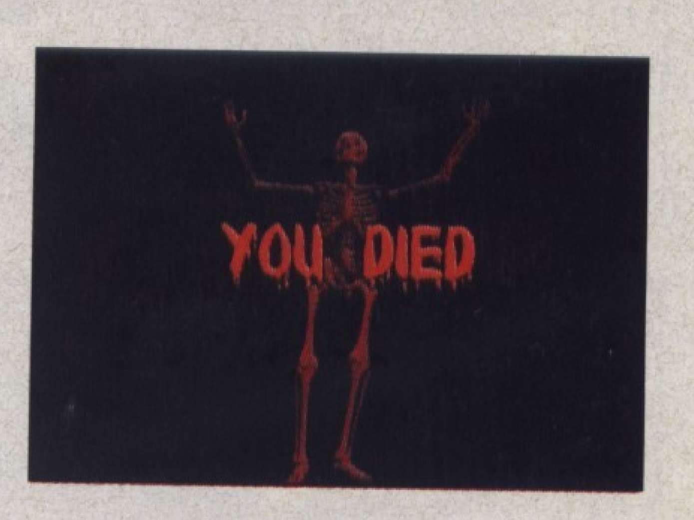
ViTAL-input 1.0

Os ofrecemos en exclusiva un programa del que se está hablando mucho y bien en el entorno DIV, creado por Vital. Crear programas que permitan la captura de los caracteres del teclado ya es bastante fácil con la rutina ViTAL-input. Hasta la actualidad esto sólo era posible en DiV2 mediante la invocación de scan_code, no obstante sus rangos de velocidad en ordenadores lentos hacían imposible este cometido. Podéis encontrar en el CD de la revista los archivos correspondientes al código fuente, así

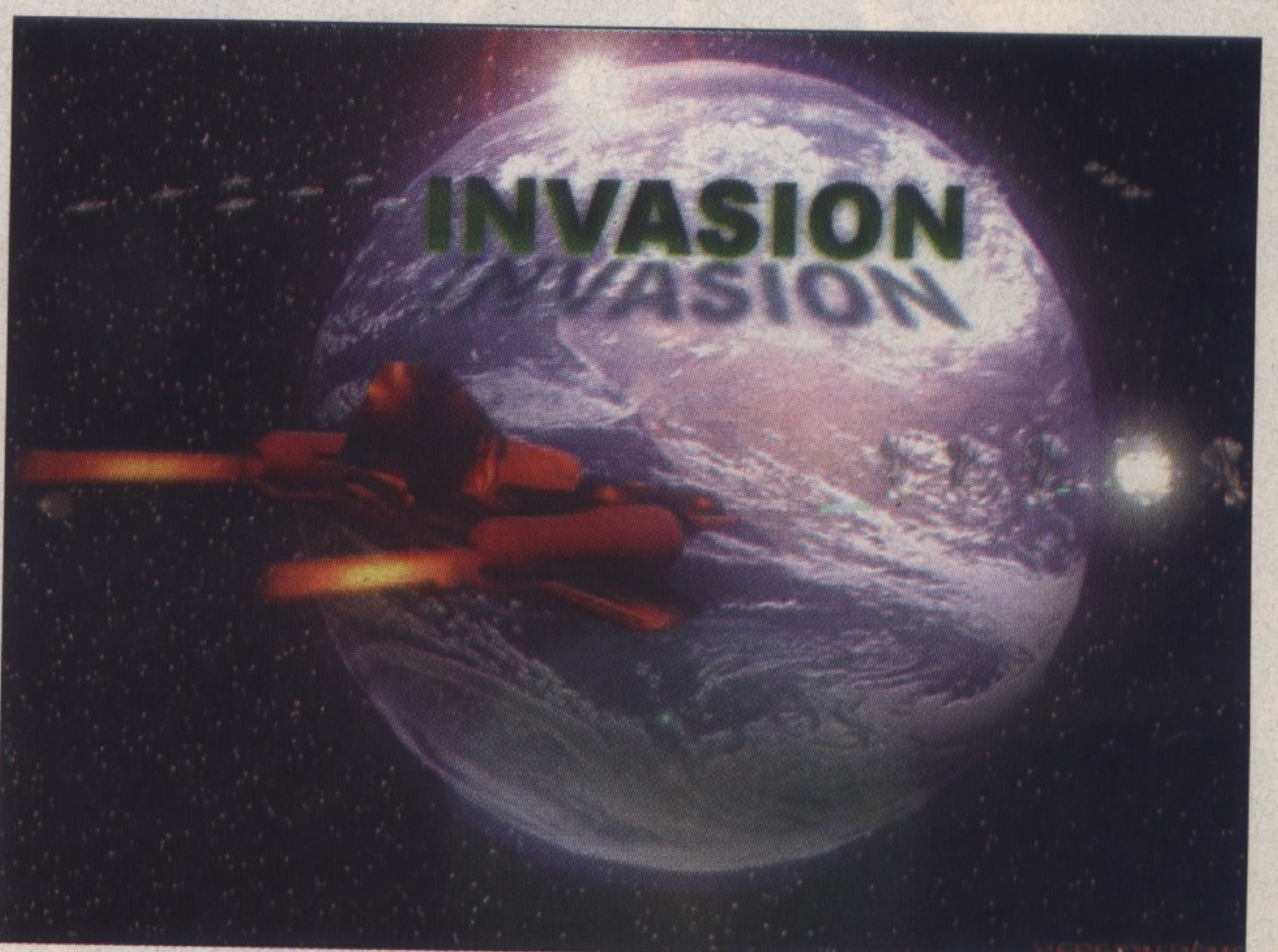










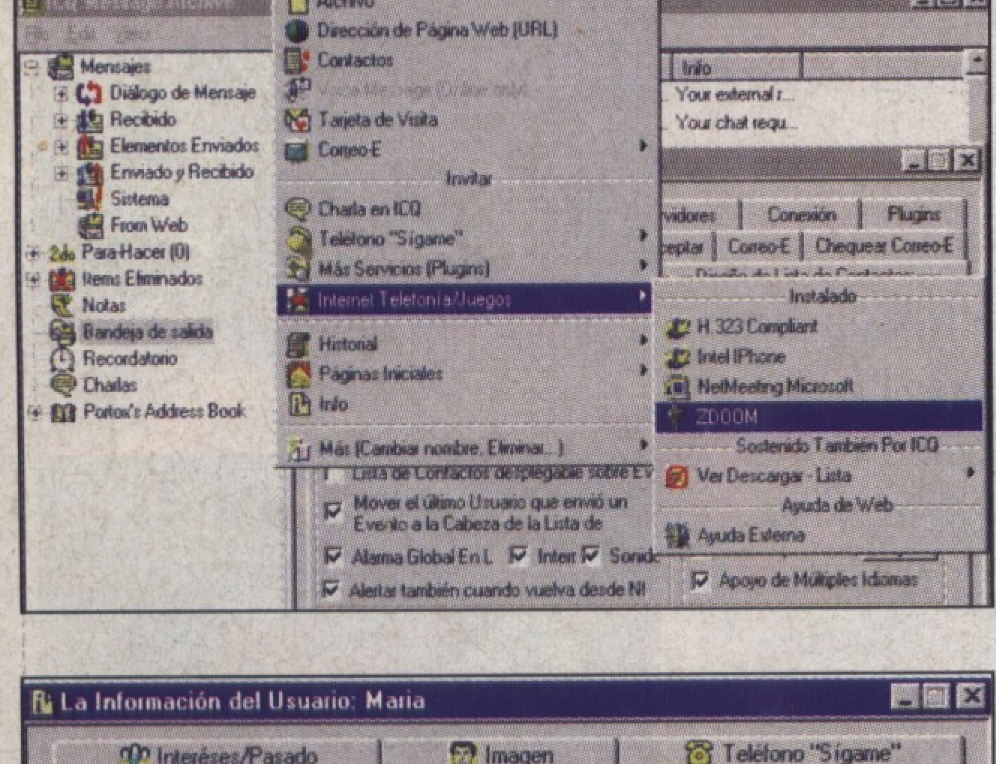


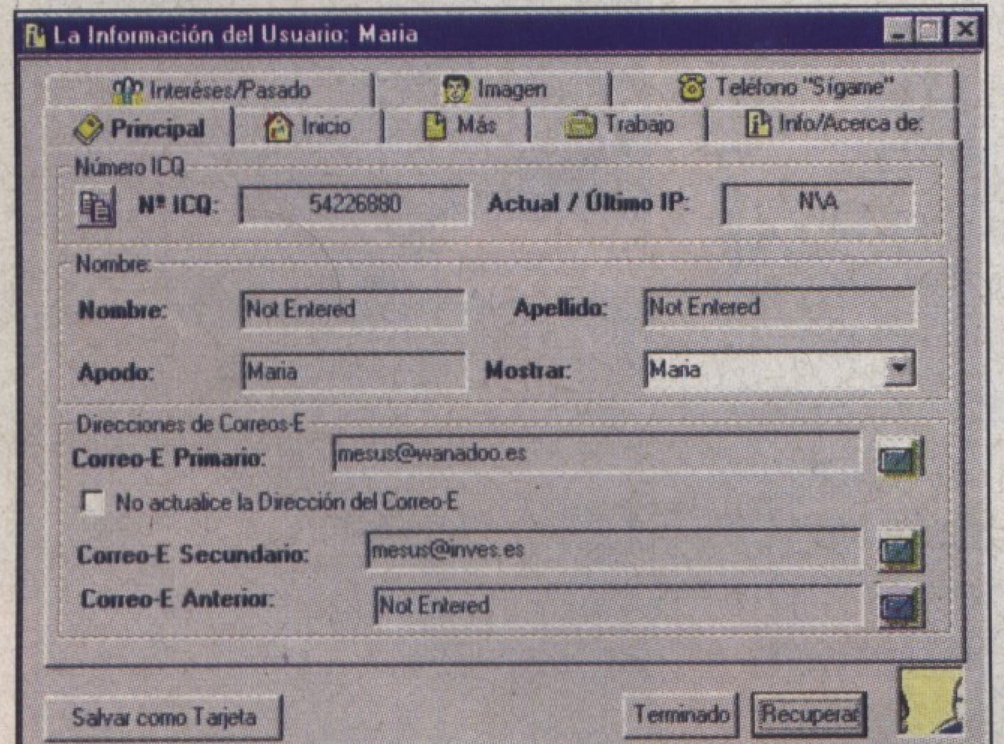


como un ejecutable para que podáis probar la rutina incluso desde el propio CD.

LingoWare

Es un programa creado básicamente para traducir el ICQ al Español, aunque también puede traducir otros programas, su funcionamiento es bastante sencillo y gracias a este programa conoceremos todas y cada una de las interesantes funciones que incluye el ICQ.





DJ´s Project 2000

Creado por Pol Jeremías, miembro de Analizer Studios. El programa tiene poco que ver con DIV pero sí con programación (Visual Basic, API win...) y su utilidad es manifiesta. Se trata de un programa para "hacerse DiscJockey". El programa permite mezclar desde mp3 hasta CD pasando y Wav, Mid. Además de poder reproducir a la vez las canciones que quieras puedes que el mismo se ocupe de mezclar canciones aplicar Pitch.

Corrector ortográfico

Creado por Anthony Rostron, es un corrector creado por un foráneo que lleva varios años metido en el mundo de la programación y deseaba un programa que corrigiera sus, en principio, múltiples errores ortográficos. Ahora incluye funciones para corregir el código fuente de DIV y otras lenguas de programación. El programa nos puede dar sugerencias y corregir nuestros errores. Cuando analizamos un archivo *.PRG de DIV el programa guardará un informe con todas las líneas que tienen texto y todas las líneas sospechosas. Se adjunta el informe generado para el primer juego premiado de Divmanía 6, "Tcraft.prg" que contenía numerosos errores.

DIVRC

En nuestras "Cartas al director" has podido leer un texto sobre la moral del programador que, como se comentaba, se apoyaba en unos archivos, incluido un programa sobre optimización que encontrarás en esta carpeta.

Entorno Operativo

Abraham Barreto nos envía un entorno operativo en línea de comando escrito en Q-Basic. Funciona como disco de inicio. Contiene instalación y ayuda (archivos *.bat).

Detección del sistema

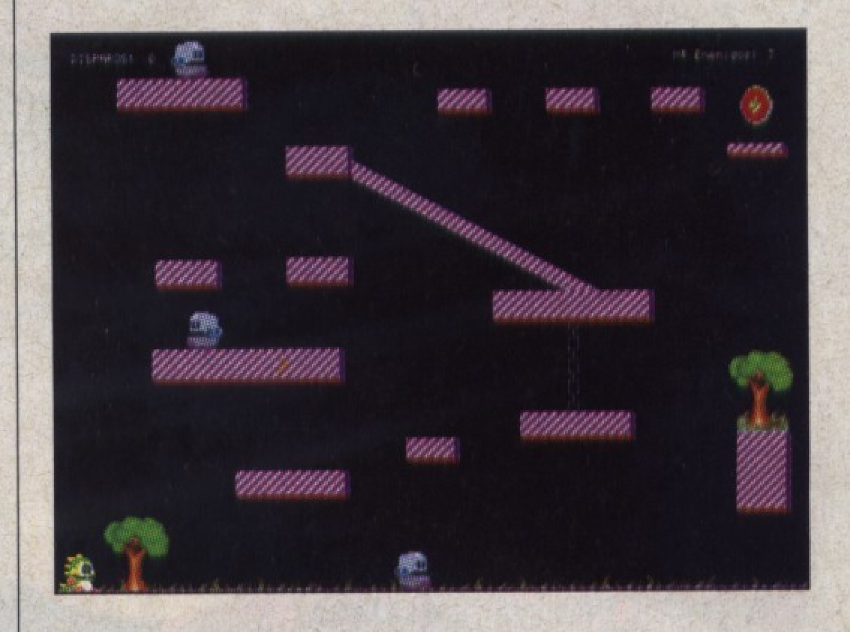
Un pequeño programa enviado por Víctor Román que puede ser útil para detectar la configuración de nuestro sistema.

Salvapantallas

Os ofrecemos un salvapantallas creado en DIV por Ismael Aguilera.

Juego: Bubble

Para acabar, una versión divera de un famoso juego: *Bubble bubble*. Para que paséis un rato muy divertido.



REVISTAS ELECTRÓNICAS

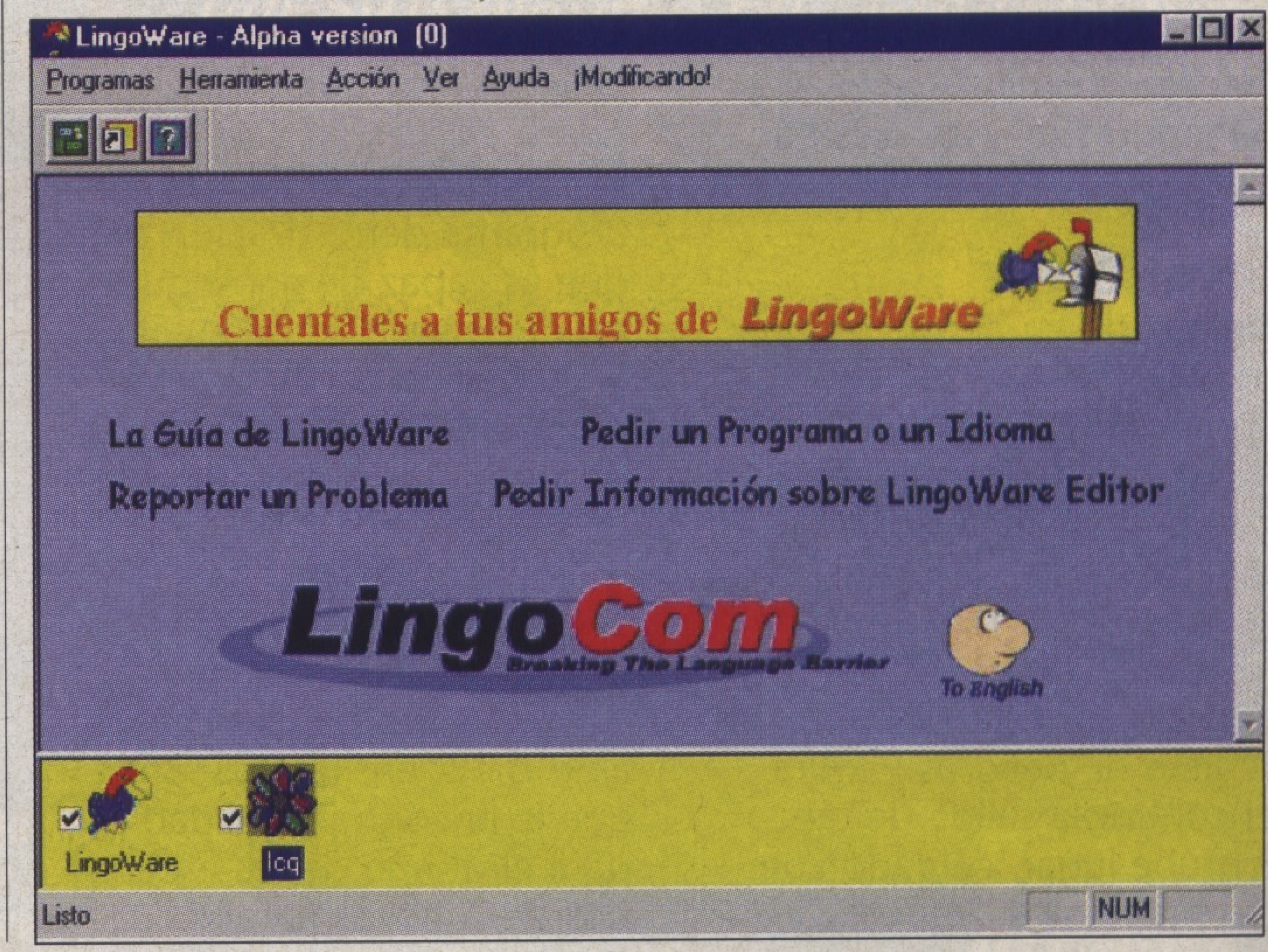
Divnet y Divworld

Últimas actualizaciones de dos revistas sobre DIV que puedes encontrar en la Red, aquí no te hará falta conectarte a Internet.



ARTÍCULOS

Todo lo que necesitas para entender a fondo los artículos de la revista.



CONTENIDO DEL CD ROM

HERRAMIENTAS PARA TUS JUEGOS

Ponemos a disposición de los lectores de DIVmanía tres demos de programas de actualidad, los juegos ganadores y un montón de cosas de utilidad.

DIVmanía
NÚMERO 8
DEPÓSITO LEGAL: M-42077-1998
• Demos
• Painter 3D
• Poser
• Canoma
• Juegos ganadores
Invasión, Dark Castle, Laberin
• Programas de regalo
Vital Input 1.0, Lingoware, DJ´s Project 2000, Corrector ortográfico, DIVRC, Entorno Operativo, Detección del sistema, Salvapantallas, Juego: Bubble
• Revistas electrónicas
Divnet y Divworld
• Artículos
Completamos los artículos de las secciones de DIVmanía
• Librerías
• Shareware
PC CD rom
Prens Técnic@

DEMOS

Painter 3D
Esta demo de este conocido programa os facilitará la labor de texturizar vuestros objetos en tres dimensiones, lucirán con vida propia en vuestros juegos.

Poser 4
Programa demostración de lo que es capaz de conseguir esta herramienta de creación y animación de personajes, un complemento a la nueva sección de la revista basada en *Poser*.

Canoma
La utilidad de *Canoma* es convertir una simple foto en algo tridimensional. Seguro que le encontráis utilidad para el diseño de los escenarios de vuestros juegos.

JUEGOS GANADORES
En este mes los afortunados ganadores en reñida lid han sido: Invasion, Dark Castle y Laberyn.

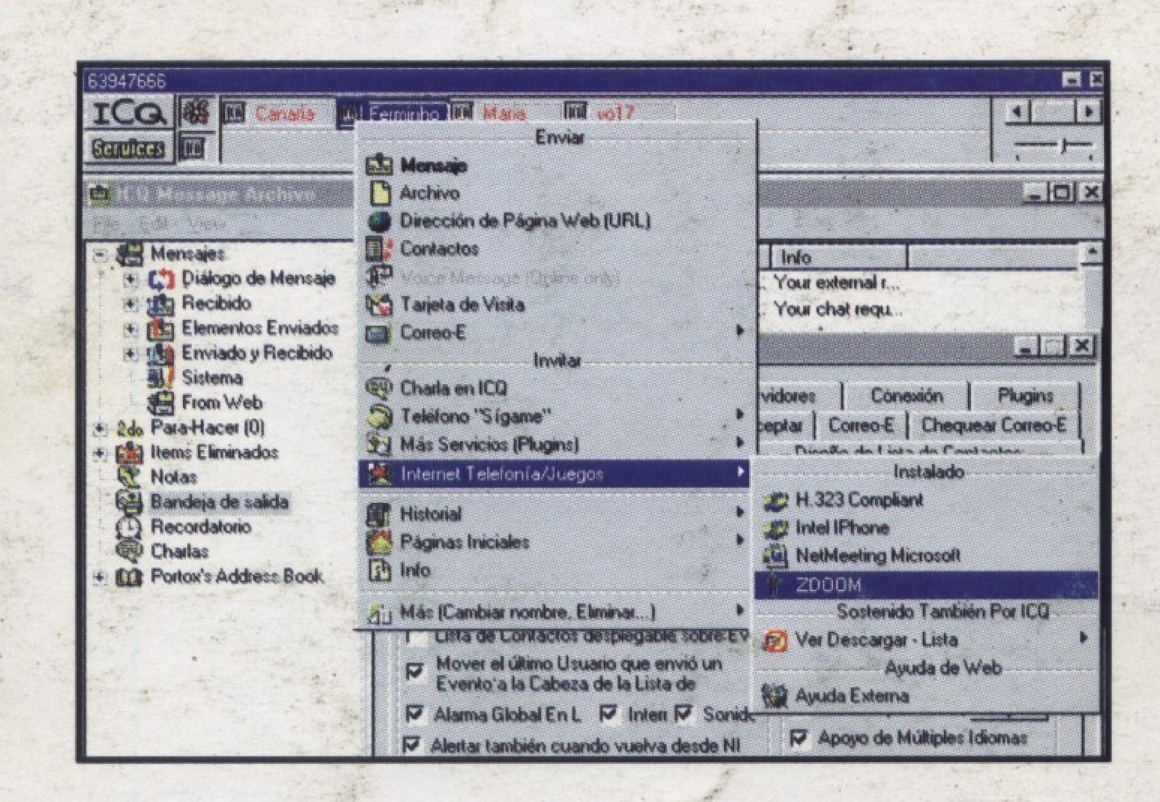
PROGRAMAS
Diversas utilidades, creadas por los lectores que os harán el trabajo más fácil: ViTAL-input 1.0, LingoWare, DJ´s Project 2000, Corrector ortográfico, DIVRC, un entorno operativo, detectores de sistema, salvapantallas y el juego Bubble.

REVISTAS ELECTRÓNICAS
Los nuevos números de Divnet y Divworld.

ACCIÓN PLATAFORMAS. Nueva sección para aprender a programar juegos de plataformas.



PROGRAMAS. Herramientas para diseñar los escenarios para vuestros juegos.



CONCURSO DE JUEGOS. Os presentamos a los ganadores de nuestro reñido concurso de videojuegos.

